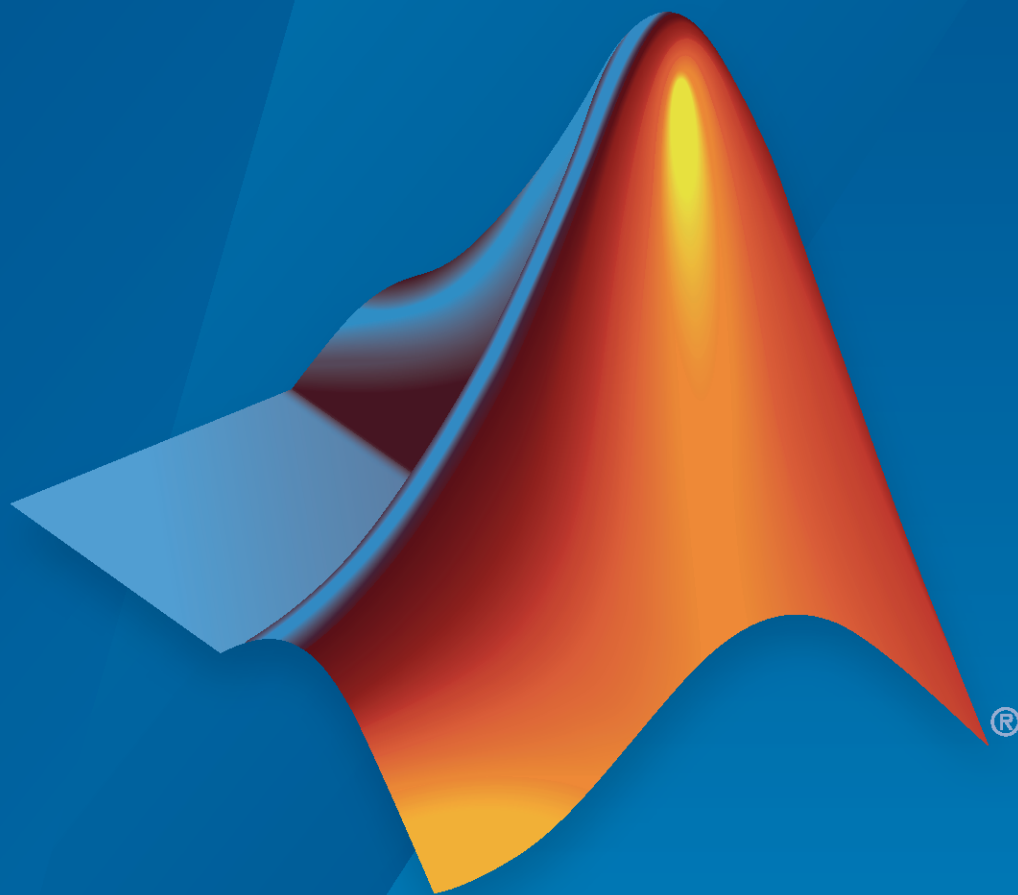


Polyspace® Bug Finder™

User's Guide



R2023a

How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Polyspace® Bug Finder™ User's Guide

© COPYRIGHT 2013–2023 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

September 2013	Online only	New for Version 1.0 (Release 2013b)
March 2014	Online Only	Revised for Version 1.1 (Release 2014a)
October 2014	Online Only	Revised for Version 1.2 (Release 2014b)
March 2015	Online Only	Revised for Version 1.3 (Release 2015a)
September 2015	Online Only	Revised for Version 2.0 (Release 2015b)
October 2015	Online Only	Rereleased for Version 1.3.1 (Release 2015aSP1)
March 2016	Online Only	Revised for Version 2.1 (Release 2016a)
September 2016	Online Only	Revised for Version 2.2 (Release 2016b)
March 2017	Online Only	Revised for Version 2.3 (Release 2017a)
September 2017	Online Only	Revised for Version 2.4 (Release 2017b)
March 2018	Online Only	Revised for Version 2.5 (Release 2018a)
September 2018	Online Only	Revised for Version 2.6 (Release 2018b)
March 2019	Online Only	Revised for Version 3.0 (Release 2019a)
September 2019	Online Only	Revised for Version 3.1 (Release 2019b)
March 2020	Online Only	Revised for Version 3.2 (Release 2020a)
September 2020	Online Only	Revised for Version 3.3 (Release 2020b)
March 2021	Online Only	Revised for Version 3.4 (Release 2021a)
September 2021	Online Only	Revised for Polyspace Bug Finder Version 3.5, Polyspace Bug Finder Server Version 3.5, and Polyspace Bug Finder Access Version 3.1 (Release 2021b)
March 2022	Online Only	Revised for Polyspace Bug Finder Version 3.6, Polyspace Bug Finder Server Version 3.6, and Polyspace Access Version 4.0 (Release 2022a)
September 2022	Online Only	Revised for Polyspace Bug Finder Version 3.7, Polyspace Bug Finder Server Version 3.7, and Polyspace Access Version 4.1 (Release 2022b)
March 2023	Online Only	Revised for Polyspace Bug Finder Version 3.8, Polyspace Bug Finder Server Version 3.8, and Polyspace Access Version 4.2 (Release 2023a)

Introduction

1

About This User's Guide	1-2
--------------------------------------	------------

Configure Analysis on Desktop

2

Set Up Polyspace Projects on Desktop

Add Source Files for Analysis in Polyspace Desktop User Interface	2-2
Polyspace Project and Source File Paths	2-2
Add Sources from Build Command	2-3
Add Sources Manually	2-5
Add Source Files Based on AUTOSAR Design Specifications	2-6
Contents of Polyspace Project and Results Folders	2-7
File Organization	2-7
Files in the Results Folder	2-7
Create Polyspace Projects from Visual Studio Build	2-9
Create Polyspace Project from Build in Visual Studio Developer Command Prompt	2-9
Create Polyspace Project from Build in Visual Studio IDE	2-10
Create Project in Polyspace Desktop User Interface Using Configuration Template	2-13
Why Use Templates	2-13
Use Predefined Template	2-13
Create Your Own Template	2-13
Sharing Project Templates	2-15
Update Project in Polyspace Desktop User Interface	2-17
Change Folder Path	2-17
Refresh Source List	2-18
Refresh Project Created from Build Command	2-18
Add Source and Include Folders	2-18
Manage Include File Sequence	2-18

Organize Layout of Polyspace Desktop User Interface	2-20
Create Your Own Layout	2-20
Save and Reset Layout	2-21
Customize Polyspace Desktop User Interface	2-22
Possible Customizations	2-22
Storage of Polyspace User Interface Customizations	2-24
Upload Results to Polyspace Access	2-25
Upload Results from Polyspace Desktop Client	2-25
Upload Results at Command Line	2-26
Results Upload Compatibility and Permissions	2-26

Run Polyspace Analysis on Desktop

3

Run Analysis in Polyspace Desktop User Interface	3-2
Arrange Layout of Windows for Project Setup	3-2
Set Product and Result Location	3-3
Start and Monitor Analysis	3-4
Fix Compilation Errors	3-4
Open Results	3-4
Storage of Temporary Files During Polyspace Analysis	3-6

Run Polyspace Analysis with Windows or Linux Scripts

4

Run Polyspace Analysis from Command Line	4-2
Specify Sources and Analysis Options Directly	4-2
Specify Sources and Analysis Options in Text File	4-2
Create Options File from Build System	4-3
Modularize Polyspace Analysis by Using Build Command	4-5
Build Source Code	4-5
Create One Polyspace Options File for Full Build	4-7
Create Options File for Specific Binary in Build Command	4-8
Create One Options File Per Binary Created in Build Command	4-8
Select Files for Polyspace Analysis Using Pattern Matching	4-11
When to Specify File Selection Patterns	4-11
Supported Patterns for File Selection	4-12
Configure Polyspace Analysis Options in User Interface and Generate Scripts	4-15
Prerequisites	4-16
Generate Scripts from Configuration	4-16
Run Analysis with Generated Scripts	4-17

Run Polyspace Analysis with MATLAB Scripts

5

Integrate Polyspace with MATLAB and Simulink	5-2
Same Release of Polyspace and MATLAB	5-2
MATLAB Release Earlier Than Polyspace	5-3
Check Integration Between MATLAB and Polyspace	5-4
Get Started with Polyspace Analysis by Using MATLAB	5-5
Prerequisites	5-5
Run Polyspace Analysis by Using MATLAB	5-5
Frequently Used MATLAB Functions	5-6
Run Polyspace Analysis by Using MATLAB Scripts	5-9
Prerequisites	5-9
Specify Multiple Source Files	5-9
Check for MISRA C:2012 Violations	5-10
Check for Specific Defects or Coding Rule Violations	5-10
Find Files That Do Not Compile	5-11
Run Analysis on Server	5-11
Compare Results from Different Polyspace Runs by Using MATLAB Scripts	5-13
Review Only New Results Compared to Last Run	5-13
Review New Results and Unreviewed Results from Last Run	5-14
Generate MATLAB Scripts from Polyspace User Interface	5-16
Prerequisites	5-16
Create Scripts from Polyspace Projects	5-16
Troubleshoot Polyspace Analysis from MATLAB	5-18
Prerequisites	5-18
Capture Polyspace Analysis Errors in Error Log	5-18

Run Polyspace Analysis in Simulink

6

Run Polyspace Analysis on Code Generated with Embedded Coder	6-2
Prerequisites	6-2
Generate and Analyze Code	6-2
Review Analysis Results	6-4
Annotate Blocks to Justify Issues	6-5
Address Polyspace Results by Annotating Simulink Blocks	6-6
Annotate Blocks Through Polyspace User Interface	6-6
Annotate Blocks in Simulink Editor	6-8
Changes in Polyspace Analysis Workflows in Simulink in R2019b ..	6-9
Code Verification Workflow in a Nutshell	6-9
Locate Pre-R2019b Menu Items in Simulink Toolstrip	6-9

Run Polyspace on Code Generated by Using Previous Releases of Simulink	6-12
Prerequisite	6-12
Run a Cross-Release Polyspace Analysis	6-12
Review Results	6-13
Run Polyspace Analysis on Code Generated from Simulink Model	6-15
Prerequisites	6-15
Open Simulink Model for Polyspace Analysis	6-15
Check for Run-Time Errors in Generated Code	6-16
Review Analysis Results	6-17
Trace and Fix Issues in the Model	6-17
Check for Coding Rule Violations	6-22
Annotate Blocks to Justify Results	6-23
Fix Model Design Issues Found as Run-time Errors and Coding Rule Violations in Generated Code	6-25
Prerequisites	6-25
Open Model	6-25
Detect and Fix Run-Time Errors	6-25
Detect and Fix Coding Rule Violations	6-27
Run Polyspace Analysis on Generated Code by Using Packaged Options Files	6-29
Generate and Package Polyspace Options Files	6-29
Run Polyspace Analysis by Using the Packaged Options Files	6-31
Run Polyspace Analysis on Custom Code in Simulink Models	6-32
Prerequisite	6-32
Analyze Custom Code	6-32
Review Analysis Results	6-33
Run Polyspace Analysis on S-Function Code	6-35
Prerequisites	6-35
S-Function Analysis Workflow	6-35
Compile S-Functions to Be Compatible with Polyspace	6-35
Example S-Function Analysis	6-35
Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts	6-37
Prerequisites	6-37
C/C++ Function Called Once in Model	6-37
C/C++ Function Called Multiple Times in Model	6-40
Run Polyspace Analysis on Custom Code in C Function Block	6-45
Prerequisites	6-45
Open Model for Running Polyspace Analysis on Custom Code in C Function Block	6-45
Run Polyspace Analysis	6-46
Identify Issues in C Code	6-47
Fix Identified Issues	6-49
Recommended Model Configuration Parameters for Polyspace Analysis	6-51

Configure Polyspace Options in Simulink	6-53
Configure Options	6-53
Share and Reuse Configuration	6-55
How Polyspace Analysis of Generated Code Works	6-56
Default Polyspace Options for Code Generated with Embedded Coder	6-57
Default Options	6-57
Constraint Specification	6-57
Recommended Polyspace options for Verifying Generated Code ...	6-58
Hardware Mapping Between Simulink and Polyspace	6-58
External Constraints on Polyspace Analysis of Generated Code ...	6-59
Extract External Constraints from Model	6-59
Storage Classes Supported for Constraint Extraction From Simulink Model	6-60
Specify Custom External Constraints	6-60
Run Polyspace Analysis on Code Generated with TargetLink	6-62
Configure and Run Analysis	6-62
Review Analysis Results	6-63
Default Polyspace Options for Code Generated with TargetLink ...	6-64
TargetLink Support	6-64
Default Options	6-64
Lookup Tables	6-64
Data Range Specification	6-65
Code Generation Options	6-65
Troubleshoot Navigation from Code to Model	6-66
Links from Code to Model Do Not Appear	6-66
Links from Code to Model Do Not Work	6-66
Your Model Already Uses Highlighting	6-67
Polyspace Support of MATLAB and Simulink from Different Releases	6-68
Complete Integration	6-70
Cross-Release Integration	6-70
Partial Integration	6-71
Navigate Back to Model	6-71

Run Polyspace Analysis in MATLAB Coder

7

Run Polyspace on C/C++ Code Generated from MATLAB Code	7-2
Prerequisites	7-2
Run Polyspace Analysis	7-2
Review Analysis Results	7-4
Run Analysis for Specific Design Range	7-5

Configure Advanced Polyspace Options in MATLAB Coder App	7-7
Configure Options	7-7
Share and Reuse Configuration	7-8

Configure Analysis on Servers

8

Run Polyspace Analysis on Servers

Run Polyspace Bug Finder on Server and Upload Results to Polyspace Access Web Interface	8-2
Prerequisites	8-2
Check Polyspace Installation	8-3
Run Bug Finder on Sample Files	8-3
Sample Scripts for Bug Finder Analysis on Servers	8-5
Specify Sources and Options in Separate Files from Launching Scripts	8-5
Complete Workflow	8-6
Send Email Notifications with Polyspace Bug Finder Server Results	8-8
Creating E-mail Notifications	8-8
Prerequisites	8-9
Export Results for E-mail Attachments	8-10
Assign Owners and Export Assigned Results	8-10
Offload Polyspace Analysis from Continuous Integration Server to Another Server	8-12
Install Products	8-12
Configure and Start Job Scheduler Services on Head Node and Worker Node	8-14
Offload Analysis from Client Node	8-15
Sample Scripts for Polyspace Analysis with Jenkins	8-17
Extending Sample Scripts to Your Development Process	8-17
Prerequisites	8-18
Set Up Polyspace Plugin in Jenkins	8-19
Script to Run Bug Finder, Upload Results and Send Common Notification	8-22
Script to Run Bug Finder, Upload Results and Send Personalized Notification	8-24
Sample Jenkins Pipeline Scripts for Polyspace Analysis	8-31
Prerequisites	8-31
Run Polyspace Analysis in Stages in a Pipeline Script	8-31
Integrate Polyspace Server Products with MATLAB	8-33
Integrate Polyspace Server Products with MATLAB	8-33
Check Integration Between MATLAB and Polyspace	8-33
Run Polyspace Server Products with MATLAB Scripts	8-34

Configure Job Submissions from Desktop to Server

Offload Polyspace Analysis to Remote Servers from Desktop

9

Send Polyspace Analysis from Desktop to Remote Servers	9-2
Client-Server Workflow for Running Analysis	9-2
Prerequisites	9-3
Offload Analysis in Polyspace User Interface	9-3
Send Polyspace Analysis from Desktop to Remote Servers Using Scripts	9-5
Client-Server Workflow for Running Analysis	9-5
Prerequisites	9-6
Run Remote Analysis	9-6
Manage Remote Analysis	9-7
Sample Scripts for Remote Analysis	9-9

Configure Analysis in IDEs

Run Polyspace Analysis in IDE Plugins

10

Run Polyspace Analysis on Eclipse Projects	10-2
Configure and Run Analysis	10-3
Review Analysis Results	10-5
Specify Polyspace Compiler Options Through Eclipse Project	10-7
Eclipse Refers Directly to Your Compilation Toolchain	10-7
Eclipse Uses Your Compilation Toolchain Through Build Command	10-8

Configure Polyspace as You Code

11

Configure Polyspace as You Code Extension in Visual Studio	11-2
General Settings	11-2
Polyspace Properties for Project	11-3
Configure Polyspace as You Code Extension in Visual Studio Code	11-8
Analysis Engine	11-8

Analysis Behavior On Save	11-9
Analysis Setup	11-9
Baseline	11-14
Justification Catalog	11-15
Other Settings	11-16
Configure Polyspace as You Code for Remote Development	11-17
Configure Polyspace as You Code Plugin in Eclipse	11-19
Polyspace as You Code Node	11-19
Analysis Node	11-20
Baseline Node	11-25
Generate Build Options for Polyspace as You Code Analysis in Visual Studio	11-27
Configure Polyspace as You Code to Extract Build Configuration ..	11-27
Specify Analysis Options Manually	11-29
Import Analysis Options from Polyspace Desktop Project	11-30
Generate Build Options for Polyspace as You Code Analysis in Visual Studio Code	11-32
Configure Polyspace as You Code to Extract Build Configuration ..	11-32
Specify Analysis Options Manually	11-35
Import Analysis Options from Polyspace Desktop Project	11-35
Generate Build Options for Polyspace as You Code Analysis in Eclipse	11-37
Configure Polyspace as You Code to Extract Build Configuration ..	11-37
Specify Analysis Options Manually	11-39
Import Analysis Options from Polyspace Desktop Project	11-39
Generate Build Options for Polyspace as You Code Analysis at the Command Line	11-41
Use polyspace-configure to Generate Build Options File	11-41
Specify Analysis Options Manually	11-42
Import Analysis Options from Polyspace Desktop Project	11-43
Baseline Polyspace as You Code Results in Visual Studio	11-44
What Baselined Results Look Like	11-44
Baselining Steps	11-45
Baseline Polyspace as You Code Results in Visual Studio Code ..	11-48
What Baselined Results Look Like	11-48
Baselining Steps	11-49
Baseline Polyspace as You Code Results in Eclipse	11-53
What Baselined Results Look Like	11-53
Baselining Steps	11-54
Baseline Polyspace as You Code Results on Command Line	11-56
What Baselined Results Look Like	11-56
Baselining Steps	11-57
Step 1: Identify Project to Use as Baseline	11-57
Step 2: Download Baseline	11-58
Step 3: Use Baseline	11-58

Configure Checkers for Polyspace as You Code in Eclipse	11-60
Select Checkers and Coding Rules	11-60
Modify Checker Behavior	11-62
Configure Checkers for Polyspace as You Code in Visual Studio ..	11-63
Select Checkers and Coding Rules	11-63
Modify Checker Behavior	11-65
Configure Checkers for Polyspace as You Code in Visual Studio Code	
.....	11-66
Configure Checkers in Checkers File	11-66
Modify Checkers Behavior	11-68
Configure Checkers for Polyspace as You Code at the Command Line	
.....	11-70
Configure Checkers and Coding Rules Directly at the Command Line	
.....	11-70
Configure Checkers in Checkers file	11-71
Modify Checkers Behavior	11-73
Analysis Scope of Polyspace as You Code	11-75
Results Involve Current File Only	11-75
Headers Included in Current File Not Analyzed	11-75
Checkers Deactivated in Polyspace as You Code Analysis	11-78
Checkers and Coding Rule Deactivated in Polyspace as You Code	
.....	11-78
Checkers with Reduced Scope in Polyspace as You Code	11-80
Troubleshoot Failed Analysis or Unexpected Results in Polyspace as You Code	
.....	11-81
Issue	11-81
Possible Solutions	11-81

Configuration Workflows Common to All Platforms

12

Configure Polyspace Analysis

Specify Polyspace Analysis Options	12-2
Polyspace User Interface	12-2
Windows or Linux Scripts	12-2
MATLAB Scripts	12-3
Eclipse and Eclipse-based IDEs	12-3
Simulink	12-3
MATLAB Coder App	12-3
Options Files for Polyspace Analysis	12-5
What are Options Files	12-5

Specifying Options Files	12-5
Specifying Multiple Options Files	12-6

Configure Target and Compiler Options

13

Specify Target Environment and Compiler Behavior	13-2
Extract Options from Build Command	13-2
Specify Options Explicitly	13-3
C/C++ Language Standard Used in Polyspace Analysis	13-5
Supported Language Standards	13-5
Default Language Standard	13-5
C11 Language Elements Supported in Polyspace	13-8
C++11 Language Elements Supported in Polyspace	13-10
C++14 Language Elements Supported in Polyspace	13-13
C++17 Language Elements Supported in Polyspace	13-16
Provide Standard Library Headers for Polyspace Analysis	13-20
Create Polyspace Analysis Configuration from Build Command (Makefile)	13-22
Requirements for Project Creation from Build Systems	13-24
Compiler Requirements	13-24
Build Command Requirements	13-25
Supported Keil or IAR Language Extensions	13-27
Special Function Register Data Type	13-27
Keywords Removed During Preprocessing	13-28
Remove or Replace Keywords Before Compilation	13-29
Remove Unrecognized Keywords	13-29
Remove Unrecognized Function Attributes	13-31
Gather Compilation Options Efficiently	13-32

Configure Inputs and Stubbing Options

14

Specify External Constraints for Polyspace Analysis	14-2
Create Constraint Template	14-2
Create Constraint Template from Code Prover Analysis Results ...	14-3
Update Existing Template	14-4
Specify Constraints in Code	14-5

External Constraints for Polyspace Analysis	14-6
Effect of External Constraints	14-6
Constraint Specification	14-7
Constraint Specification Limitations	14-11
Constrain Global Variable Range for Polyspace Analysis	14-12
User Interface (Desktop Products Only)	14-12
Command Line	14-13
Constrain Function Inputs for Polyspace Analysis	14-14
User Interface (Desktop Products Only)	14-14
Command Line	14-15
XML File Format for Polyspace Analysis Constraints	14-17
Syntax Description — XML Elements	14-17
Valid Modes and Default Values	14-21

15

Configure Multitasking Analysis

Analyze Multitasking Programs in Polyspace	15-2
Configure Analysis	15-3
Review Analysis Results	15-4
Differences Between Bug Finder and Code Prover	15-5
Auto-Detection of Thread Creation and Critical Section in Polyspace	15-7
Multitasking Routines that Polyspace Can Detect	15-7
Example of Automatic Thread Detection	15-8
Naming Convention for Automatically Detected Threads	15-11
Limitations of Automatic Thread Detection	15-12
Configuring Polyspace Multitasking Analysis Manually	15-17
Specify Options for Multitasking Analysis	15-17
Adapt Code for Code Prover Multitasking Analysis	15-17
Protections for Shared Variables in Multitasking Code	15-21
Detect Unprotected Access	15-21
Protect Using Critical Sections	15-22
Protect Using Temporally Exclusive Tasks	15-23
Protect Using Priorities	15-23
Protect By Disabling Interrupts	15-24
Define Atomic Operations in Multitasking Code	15-25
Nonatomic Operations	15-25
What Polyspace Considers as Nonatomic	15-25
Define Specific Operations as Atomic	15-26
Define Task Priorities for Data Race Detection in Polyspace	15-28
Emulating Task Priorities	15-28
Examples of Task Priorities	15-28
Further Explorations	15-29

Effect of Task Priorities in Code Prover	15-29
Define Critical Sections with Functions That Take Arguments ...	15-31
Polyspace Assumption on Functions Defining Critical Sections ...	15-31
Adapt Polyspace Analysis to Lock and Unlock Functions with Arguments	15-31

Configure Coding Rules Checking and Code Metrics Computation

16

Check for and Review Coding Standard Violations	16-2
Configure Coding Rules Checking	16-2
Review Coding Rule Violations	16-6
Generate Reports	16-7
Avoid Violations of MISRA C:2012 Rules 8.x	16-8
Reduce Software Complexity by Using Polyspace Checkers	16-11
Configure Thresholds for Software Complexity Checkers	16-11
Identify and Reduce Software Complexity	16-12
Software Quality Objective Subsets (C:2004)	16-15
Rules in SQO-Subset1	16-15
Rules in SQO-Subset2	16-16
Software Quality Objective Subsets (AC AGC)	16-19
Rules in SQO-Subset1	16-19
Rules in SQO-Subset2	16-20
Software Quality Objective Subsets (C:2012)	16-23
Guidelines in SQO-Subset1	16-23
Guidelines in SQO-Subset2	16-24
Software Quality Objective Subsets (C++)	16-27
SQO Subset 1 - Direct Impact on Selectivity	16-27
SQO Subset 2 - Indirect Impact on Selectivity	16-28
Coding Rule Subsets Checked Early in Analysis	16-33
MISRA C:2004 and MISRA AC AGC Rules	16-33
MISRA C:2012 Rules	16-40
Create Custom Coding Rules	16-48
Specify Naming Convention	16-48
Check for Violations of Defined Custom Coding Rule	16-50
Compute Code Complexity Metrics Using Polyspace	16-51
Impose Limits on Metrics (Desktop Products Only)	16-51
Impose Limits on Metrics (Server and Access products)	16-53

HIS Code Complexity Metrics	16-54
Project	16-54
File	16-54
Function	16-54
Migrate Code Prover Workflows for Checking Coding Standards and Code Metrics to Bug Finder	16-55
Changes in Workflow	16-55

Polyspace Coverage of Coding Standards

17

Polyspace Support for Coding Standards	17-2
Summary of Polyspace Support	17-2
AUTOSAR C++14	17-2
MISRA C++:2008	17-4
MISRA C:2012	17-4
CERT C	17-7
Other	17-8
MISRA C:2004 and MISRA AC AGC Coding Rules	17-9
Supported MISRA C:2004 and MISRA AC AGC Rules	17-9
Troubleshooting	17-9
List of Supported Coding Rules	17-9
Unsupported MISRA C:2004 and MISRA AC AGC Rules	17-41
Required or Mandatory MISRA C:2012 Rules Supported by Polyspace Bug Finder	17-43
Mandatory Rules	17-43
Required Rules	17-44
Decidable MISRA C:2012 Rules Supported by Polyspace Bug Finder	17-54
Undecidable MISRA C:2012 Rules and Directives Supported by Polyspace Bug Finder	17-64
Undecidable Rules	17-64
Undecidable Directives	17-67
Polyspace Support for MISRA C: 2012 Amendments	17-69
MISRA C:2012 Technical Corrigendum 1	17-69
MISRA C: 2012 Amendment 1 (AMD1)	17-69
MISRA C:2012 Amendment 2 (AMD2)	17-71
Essential Types in MISRA C:2012 Rules 10.x	17-72
Categories of Essential Types	17-72
How MISRA C:2012 Uses Essential Types	17-72
Unsupported MISRA C:2012 Guidelines	17-74
Required and Statically Enforceable CERT C Rules Supported by Polyspace Bug Finder	17-75

Required MISRA C++:2008 Coding Rules Supported by Polyspace	
Bug Finder	17-83
Supported Rules	17-83
Unsupported Rules	17-99
JSF AV C++ Coding Rules	17-100
Supported JSF C++ Coding Rules	17-100
Unsupported JSF++ Rules	17-118
Required AUTOSAR C++14 Coding Rules Supported by Polyspace	
Bug Finder	17-125
Supported Rules	17-125
Unsupported Rules	17-153
Statically Enforceable AUTOSAR C++14 Rules Supported by	
Polyspace Bug Finder	17-156
Automated Rules	17-156
Partially Automated Rules	17-182

Configure Bug Finder Checkers

18

Choose Specific Bug Finder Defect Checkers	18-2
User Interface (Desktop Products Only)	18-2
Command Line	18-2
Modify Default Behavior of Bug Finder Checkers	18-3
Defect Checkers and Coding Rules Modified by Analysis Options ..	18-3
Modify Bug Finder Checkers Through Code Behavior Specifications	
.....	18-12
XML Format	18-12
Datalog Format	18-16
Flag Deprecated or Unsafe Functions, Keywords, or Macros Using	
Bug Finder Checkers	18-21
Identify Need for Extending Checker	18-21
Extend Checker	18-22
Checkers That Can Be Extended	18-22
Extend Bug Finder Checkers for Standard Library Functions to	
Custom Libraries	18-24
Identify Need for Extending Checker	18-24
Extend Checker	18-24
Checkers That Can Be Extended	18-25
Limitations	18-25
Extend Bug Finder Checkers to Find Defects from Specific System	
Input Values	18-26
Identify Need for Extending Checker	18-26
Extend Checker	18-26
Checkers That Can Be Extended	18-27

Extend Concurrency Defect Checkers to Unsupported Multithreading	
Environments	18-30
Identify Need for Extending Checker	18-30
Extend Checker	18-31
Checkers That Can Be Extended	18-31
Limitations	18-32
Extend Checkers for Initialization to Check Function Arguments	
Passed by Pointers	18-33
Identify Need for Existing Checker	18-33
Extend Checker	18-33
Checkers That Can Be Extended	18-34
Extend Data Race Checkers to Atomic Operations	18-35
Identify Need for Extending Checker	18-35
Extend Checker	18-35
Checkers That Can Be Extended	18-36
Prepare Checkers Configuration for Polyspace Bug Finder Analysis	
.....	18-38
Identify Checkers to Enable	18-38
Create Checkers Configuration Files	18-40
Bug Finder Defect Groups	18-43
C++ Exceptions	18-43
Concurrency	18-43
Cryptography	18-44
Data flow	18-44
Dynamic Memory	18-45
Good Practice	18-45
Numerical	18-45
Object Oriented	18-45
Performance	18-46
Programming	18-46
Resource Management	18-47
Static Memory	18-47
Security	18-47
Tainted data	18-47
Classification of Defects by Impact	18-49
High Impact Defects	18-49
Medium Impact Defects	18-51
Low Impact Defects	18-56
Sources of Tainting in a Polyspace Analysis	18-61
Sources of Tainted Data	18-61
Impact of Tainted Data Defects	18-61
Polyspace Bug Finder Defects Checkers Enabled by Default	18-65
Polyspace Bug Finder Defects Checkers Enabled by Default for Generated Code	18-70
Bug Finder Results Found in Fast Analysis Mode	18-72
Polyspace Bug Finder Defects	18-72

MISRA C:2004 and MISRA AC AGC Rules	18-75
MISRA C:2012 Rules	18-81
MISRA C++ 2008 Rules	18-87

Extend CWE Coding Standard Coverage Using Polyspace Defect Checkers	18-97
Find CWE IDs from Polyspace Results	18-97
Mapping Between CWE Identifiers and Polyspace Results	18-97

Configure File Sets for Bug Finder Analysis

19

Classify Project Files Into File Sets for Precise Control of Bug Finder Analysis	19-2
Classification File Structure Based on Analysis Requirements	19-2
Classification File Usage	19-3
Parts of Classification File	19-4

Configure Comment Import from Previous Results

20

Import Review Information from Previous Polyspace Analysis	20-2
Automatic Import from Last Analysis	20-2
Import from Another Analysis Result	20-2
Import Algorithm	20-3
View Imported Review Information That Does Not Apply	20-4
Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results	20-5
Mapping Multiple MISRA C: 2004 Annotations to the Same MISRA C: 2012 Result	20-6

Review Results in Polyspace User Interface

Interpret Polyspace Bug Finder Results

21

Interpret Bug Finder Results in Polyspace Desktop User Interface	21-2
Interpret Result Details Message	21-3
Find Root Cause of Result	21-3
Investigate the Cause of Empty Results List	21-7

Dashboard in Polyspace Desktop User Interface	21-9
Code Covered by Analysis	21-9
Defect Distribution by Impact	21-10
Defect Distribution by Category or File	21-10
Coding Rule Violations by Rule or File	21-11
Other Dashboard Features	21-12
Concurrency Modeling in Polyspace Desktop User Interface	21-13
Results List in Polyspace Desktop User Interface	21-15
Source Code in Polyspace Desktop User Interface	21-17
Examine Source Code	21-17
Expand Macros	21-18
Manage Multiple Files in Source Pane	21-20
View Code Block	21-21
Result Details in Polyspace Desktop User Interface	21-22
Call Hierarchy in Polyspace Desktop User Interface	21-24
Actions Supported on Call Hierarchy Pane	21-25
Limitations of Call Hierarchy Display in Bug Finder	21-25
Understanding Changes in Polyspace Results After Product Upgrade	21-26
Changes in Polyspace Code Prover Results	21-26
Changes in Polyspace Bug Finder Results	21-28

Fix or Comment Polyspace Results

22

Address Results in Polyspace User Interface Through Bug Fixes or Justifications	22-2
Add Review Information to Results File	22-2
Comment or Annotate in Code	22-3

Manage Results

23

Filter and Group Results in Polyspace Desktop User Interface	23-2
Filter Results	23-3
Group Results	23-7

Generate Reports from Polyspace Results	24-2
Generate Reports from User Interface	24-2
Generate Reports from Command Line	24-3
Export Polyspace Analysis Results	24-5
Export Results to Text File	24-5
Export Results to MATLAB Table	24-5
Export Results to JSON Format	24-6
View Exported Results	24-6
Export Polyspace Analysis Results to Excel by Using MATLAB Scripts	24-9
Report Result Summary and Details in One Worksheet	24-9
Control Formatting of Excel Report	24-10
Visualize Bug Finder Analysis Results in MATLAB	24-11
Export Results to MATLAB Table	24-11
Generate Graphs from Results and Include in Report	24-11
Customize Existing Bug Finder Report Template	24-15
Prerequisites	24-15
View Components of Template	24-15
Change Components of Template	24-16
Generate Report Containing MISRA C:2012 Violations, Code Metrics, and Runtime Check Results	24-20
Prerequisite	24-20
Obtain Code Metrics and Coding Rules Results by Using Bug Finder	24-20
Obtain Run Time Check and Stack Usage Results by Using Code Prover	24-21
Generate a Combined Report	24-22

Review Results on Web Browser

Interpret Bug Finder Results in Polyspace Access Web Interface	25-2
Interpret Result Details Message	25-3
Find Root Cause of Result	25-3
Investigate the Cause of Empty Results List	25-7
Dashboard in Polyspace Access Web Interface	25-9

Code Metrics Dashboard in Polyspace Access Web Interface	25-11
Quality Objectives Dashboard in Polyspace Access	25-14
Monitor Code Quality Against Software Quality Objectives	25-14
Customize Software Quality Objectives	25-16
Results List in Polyspace Access Web Interface	25-19
Source Code in Polyspace Access Web Interface	25-21
Tooltips	25-21
Examine Source Code	25-22
Expand Macros	25-23
View Code Block	25-24
Navigate from Code to Model	25-24
Result Details in Polyspace Access Web Interface	25-26
Call Hierarchy in Polyspace Access Web Interface	25-28
Configuration Settings in Polyspace Access Web Interface	25-30
Review History in Polyspace Access Web Interface	25-33
Create Bug Tracking Tool Tickets from the Polyspace Access Web Interface	25-35
Create a Ticket	25-35
Manage Existing Tickets	25-36

Fix or Comment Polyspace Results on Web Browser

26

Address Results in Polyspace Access Through Bug Fixes or Justifications	26-2
Add Review Information in Result Details pane	26-2
Comment or Annotate in Code	26-4
Import Review Information from Existing Polyspace Access Projects	26-5
Import Review Information from Source Project to Target Project in Polyspace Access	26-5
View and Select Imported Reviews	26-6
Confirm Imported Review Information	26-6
Import Review Information at the Command-Line	26-7

27

Manage Permissions and View Project Trends in Polyspace Access Web Interface	27-2
Create a Project Folder	27-2
Manage Project Permissions	27-3
View Project Trends	27-6
Filter and Sort Results in Polyspace Access Web Interface	27-8
Filter Results	27-9
Create Custom Filter Groups in Polyspace Access Web Interface	27-11
Manage Software Quality Objectives in Polyspace Access	27-13
Manage SQOs in the User Interface	27-13
Manage SQOs at the Command Line	27-14
Add Labels to Project Runs in Polyspace Access	27-16
Manage Labels in the User Interface	27-16
Manage Labels at the Command Line	27-17
Compare Results in Polyspace Access Project to Previous Runs and View Trends	27-19
Comparison Mode in the Polyspace Access Interface	27-19
Comparison Mode at the Command Line	27-21

Export Results from Polyspace Access Web Server

28

Open or Export Results from Polyspace Access	28-2
Open Polyspace Access Results in a Desktop Interface	28-2
Export Polyspace Access Results to a TSV File	28-2
Generate Report and Variables List from Polyspace Access	28-4

Review Results in IDEs

Review Results in Polyspace as You Code

29

Run Polyspace as You Code in Visual Studio and Review Results	29-2
Confirm Installation of Extension	29-2

Run Analysis on Save	29-2
Run Analysis on Demand	29-2
Review Results	29-3
Justify Results Using Code Annotations	29-4
View Help	29-4
Configure Checkers and Other Settings	29-5
Run Polyspace as You Code in Visual Studio Code and Review Results	
.....	29-6
Check Installation and Start Extension	29-6
View Extension Information in Status Bar	29-6
Open Additional Polyspace Views	29-7
Run Analysis	29-9
Review Results	29-10
View Context-Sensitive Help for Result	29-13
Configure Checkers and Other Settings	29-13
Run Polyspace as You Code in Eclipse and Review Results	29-15
Check Installation and Start Plugin	29-15
Open Polyspace as You Code Perspective	29-15
Run Analysis	29-18
Review Results	29-18
Justify Results Using Code Annotations	29-19
View Context-Sensitive Help for Result	29-19
Configure Checkers and Other Settings	29-19
Run Polyspace as You Code from Command Line and Export Results	
.....	29-21
Add Install Folder to Path	29-21
Run Analysis and See Results on Console	29-21
Store Results in Specific Folder	29-21
Export Results to JSON Format (SARIF Output)	29-22
Specify Analysis Options by Using Options Files	29-22
Create Options File by Analyzing Build	29-22
Integrate Polyspace as You Code in IDEs and Editors Without Plugins	
.....	29-24
Overview of Approach	29-24
Integration Steps	29-24
Further Exploration	29-26
Use a Justification Catalog to Autocomplete Annotations in Polyspace as You Code plugins	29-27
Create and Edit Justification Catalog	29-27

Review Workflows Common to All Platforms

Hide Known or Acceptable Results Using Code Annotations

30

Annotate Code and Hide Known or Acceptable Results	30-2
Code Annotation Syntax	30-2
Syntax Examples	30-6
Code Annotation Warnings	30-9
Ignoring Code Annotations	30-9
Short Names of Bug Finder Defect Groups and Defect Checkers	30-11
Bug Finder Defect Groups Short Names	30-11
Bug Finder Defect Checkers Short Names	30-11
Short Names of Code Complexity Metrics	30-24
Project Metrics	30-24
File Metrics	30-24
Function Metrics	30-24
Annotate Code for Known or Acceptable Results (Not Recommended)	30-26
Add Annotations from the Polyspace Interface	30-26
Add Annotations Manually	30-27
Define Custom Annotation Format	30-30
Define Annotation Syntax Format	30-32
Map Your Annotation to the Polyspace Annotation Syntax	30-35
Define Multiple Custom Annotation Syntaxes	30-36
Annotation Description Full XML Template	30-38
Example	30-41

Advanced Review Workflows

31

Evaluate Polyspace Bug Finder Results Against Bug Finder Quality Objectives	31-2
Comparing Analysis Results Against Quality Objectives	31-5
Justify Coding Rule Violations Using Code Prover Checks	31-7
Rules About Data Type Conversions	31-7
Rules About Pointer Arithmetic	31-9
Polyspace Results in Lines Containing Macros	31-12
Macros in Source Lines Can Be Expanded in Place	31-12

Results in Function-Like Macros Shown Only Once	31-12
Migrate Results from Polyspace Metrics to Polyspace Access . . .	31-14
Requirements for Migration	31-15
Migration of Results	31-16
Differences in SQO Between Polyspace Metrics and Polyspace Access	31-17

Troubleshooting

32 Troubleshooting in Polyspace Bug Finder

Fix License Error -4,0 When Running Polyspace	32-3
Issue	32-3
Possible Cause: Another Polyspace Instance Running	32-3
Possible Cause: Prior Polyspace Run in Simulink or MATLAB Coder	32-3
View Error Information When Analysis Stops	32-4
View Error Information in User Interface	32-4
View Error Information in Log File	32-4
Contact Technical Support About Issues with Running Polyspace	32-6
Provide System Information	32-6
Provide Information About the Issue	32-7
Provide Polyspace Analysis Statistics File (Optional)	32-8
Resolve Error: No Compilation Unit Detected in Your Build	32-9
Issue	32-9
Possible Solutions	32-9
Create Polyspace Projects from Build Systems That Use Unsupported Compilers	32-11
Issue	32-11
Cause	32-11
Solution	32-11
Fix Slow Build Process When Polyspace Traces Build	32-17
Issue	32-17
Cause	32-17
Solution	32-17
Check if Polyspace Supports Build Scripts	32-18
Issue	32-18
Possible Cause	32-18
Solution	32-18

Troubleshoot Project Creation from MinGW Build	32-19
Issue	32-19
Cause	32-19
Solution	32-19
Troubleshoot Project Creation from Visual Studio Build	32-20
Fix Error: Polyspace Cannot Find Server	32-21
Message	32-21
Possible Cause	32-21
Solution	32-21
Fix Error: Job Manager Cannot Write to Database	32-22
Message	32-22
Possible Cause	32-22
Workaround	32-22
Fix Polyspace Compilation Errors About Undefined Identifiers ..	32-23
Issue	32-23
Possible Cause: Missing Files	32-23
Possible Cause: Unrecognized Keyword	32-23
Possible Cause: Declaration Embedded in #ifdef Statements	32-24
Possible Cause: Project Created from Non-Debug Build	32-24
Fix Polyspace Compilation Errors About Unknown Function Prototype	32-26
Issue	32-26
Cause	32-26
Solution	32-26
Fix Polyspace Compilation Errors Related to #error Directive ...	32-27
Issue	32-27
Cause	32-27
Solution	32-27
Fix Polyspace Compilation Errors About Large Objects	32-28
Issue	32-28
Cause	32-28
Solution	32-28
Fix Polyspace Compilation Errors Related to Generic Compiler ..	32-30
Issue	32-30
Cause	32-30
Solution	32-30
Fix Polyspace Compilation Errors Related to GNU Compiler	32-31
Issue	32-31
Cause	32-31
Solution	32-31
Fix Polyspace Compilation Errors Related to Visual Compilers ..	32-32
Import Folder	32-32
pragma Pack	32-32
C++/CLI	32-33

Fix Polyspace Compilation Errors Related to Keil or IAR Compiler	32-34
Missing Identifiers	32-34
Fix Polyspace Compilation Errors Related to Diab Compiler	32-35
Issue	32-35
Cause	32-35
Solution	32-35
Fix Polyspace Compilation Errors Related to Green Hills Compiler	32-37
Issue	32-37
Cause	32-37
Solution	32-37
Fix Polyspace Compilation Errors Related to TASKING Compiler	32-39
Issue	32-39
Cause	32-39
Solution	32-39
Fix Polyspace Compilation Errors Related to Texas Instruments Compilers	32-41
Issue	32-41
Possible Solutions	32-41
Fix Errors from Use of Polyspace Header Files	32-42
Issue	32-42
Possible Solutions	32-42
Fix Polyspace Compilation Errors About Namespace std Without Prefix	32-44
Issue	32-44
Cause	32-44
Solution	32-44
Fix Polyspace Compilation Warnings Related to Assertion or Memory Allocation Functions	32-45
Issue	32-45
Cause	32-45
Solution	32-45
Fix Polyspace Compilation Errors About In-Class Initialization (C++)	32-46
Update Eclipse Java Version for Polyspace Plug-in	32-47
Issue	32-47
Cause	32-47
Solution	32-47
Fix MATLAB Crashes Referring to Polyspace in matlabroot	32-48
Issue	32-48
Possible Solutions	32-48

Diagnose Why Coding Standard Violations Do Not Appear as Expected	32-49
Issue	32-49
Possible Solutions	32-49
Check Why a Bug Finder Defect Does Not Appear as Expected ..	32-52
Issue	32-52
Possible Solutions	32-52
Fix Insufficient Memory Errors During Polyspace Report Generation	
.....	32-55
Issue	32-55
Possible Solutions	32-55
Fix Errors or Slow Polyspace Runs from Disk Defragmentation and Anti-virus Software	32-58
Issue	32-58
Possible Cause	32-58
Solution	32-58
Fix SQLite I/O Errors on Running Polyspace	32-60
Issue	32-60
Possible Solutions	32-60
Fix Polyspace Errors Related to Temporary Files	32-61
No Access Rights	32-61
No Space Left on Device	32-61
Cannot Open Temporary File	32-61
Fix Errors Applying Custom Annotation Format for Polyspace Results	
.....	32-63
Issue	32-63
Possible Solutions	32-63
Fix Issues When when Integrating Polyspace with MATLAB and Simulink	32-65
Issue	32-65
Possible Solutions	32-65
Check Why Polyspace Functions are Unavailable in MATLAB	32-67
Issue	32-67
Possible Solution	32-67
Troubleshoot Java Incompatibility in Polyspace Plugin for Eclipse	
.....	32-68
Issue	32-68
Possible Solutions	32-68

Polyspace Access ETL and Web Server services do not start 33-2

 Issue 33-2

 Possible Cause: Hyper-V Network Configuration Cannot Resolve Local
 Host Names 33-2

Contact Technical Support About Polyspace Access Issues 33-5

Introduction

About This User's Guide

This User's Guide covers all Polyspace Bug Finder products:

- Polyspace Bug Finder
- Polyspace Bug Finder Server™
- Polyspace Access™

Depending on how you set up a Bug Finder run, you might be running an analysis from one of these locations:

- **Desktop:** If you are running an analysis and reviewing the results on your desktop, you use Polyspace Bug Finder. For desktop-specific workflows, see “Configure Analysis on Desktop” or “Review Results in Polyspace User Interface”.
- **Server:** If you are running an analysis on a server, or reviewing the results from a server run on a web browser, you use:
 - Polyspace Bug Finder Server to run the analysis.
 - Polyspace Access to host the analysis results (for review on a web browser).

For server-specific workflows, see “Configure Analysis on Servers” or “Review Results on Web Browser”.

- **IDE:** If you are running an analysis on the current file in your Integration Development Environment (IDE), you use Polyspace as You Code. Polyspace as You Code is a feature available with Polyspace Access. For IDE-specific workflows, see “Configure Analysis in IDEs” or “Review Results in IDEs”.

The Bug Finder analysis engine underlies all Bug Finder products. Chapters that do not mention a particular platform typically describe the underlying Bug Finder analysis engine and apply to all three platforms.

Configure Analysis on Desktop

Set Up Polyspace Projects on Desktop

- “Add Source Files for Analysis in Polyspace Desktop User Interface” on page 2-2
- “Contents of Polyspace Project and Results Folders” on page 2-7
- “Create Polyspace Projects from Visual Studio Build” on page 2-9
- “Create Project in Polyspace Desktop User Interface Using Configuration Template” on page 2-13
- “Update Project in Polyspace Desktop User Interface” on page 2-17
- “Organize Layout of Polyspace Desktop User Interface” on page 2-20
- “Customize Polyspace Desktop User Interface” on page 2-22
- “Upload Results to Polyspace Access” on page 2-25

Add Source Files for Analysis in Polyspace Desktop User Interface

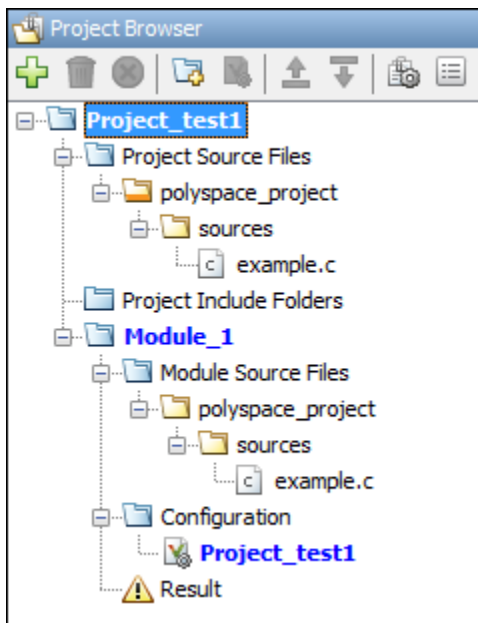
This topic shows how to create a project in the user interface of the Polyspace desktop products.

- If using the Polyspace Server products, see “Set Up Bug Finder Analysis on Servers During Continuous Integration”.
- If using Polyspace as You Code, see “Set Up Polyspace Analysis in IDEs”.

To begin a Polyspace analysis, you must specify the path to your source files and headers.

You can specify your source paths explicitly or extract them from a build command (makefile) after executing the command. If you use a build command for building your source code or build your source code in an IDE (using an underlying build command), try extracting from the build command first. If Polyspace cannot trace your build command, manually add the paths to your source and include folders. You specify the target and compiler options later. See “Target and Compiler”.

Provide the source paths in a Polyspace project. The source files are displayed on the **Project Browser** pane.



A corresponding `.psprj` file is created in the location where you saved the project. When you create a project, choose the default location for saving it or enter a new location. To change the default location, select **Tools > Preferences** and use the options on the **Project and Results Folder** tab.

Polyspace Project and Source File Paths

A Polyspace project points to source files using their absolute paths. However, each time you reopen a project in the Polyspace user interface, the absolute paths to the sources are recomputed relative to the current location of the project.

For instance, suppose that a project is stored in:

```
//networkLocation/polyspaceProjects/
```

Suppose that the project points to the source file path:

```
//networkLocation/src/file.c
```

If you move the project to

```
//usr/local/polyspaceProjects/
```

and open the project in the user interface, it now points to the source file path:

```
//usr/local/src/file.c
```

(Note that if you open the project file in a text editor, it continues to show the old path. You have to run an analysis using the newly moved project for the new paths to be hardcoded in the project and show up even in a text editor.)

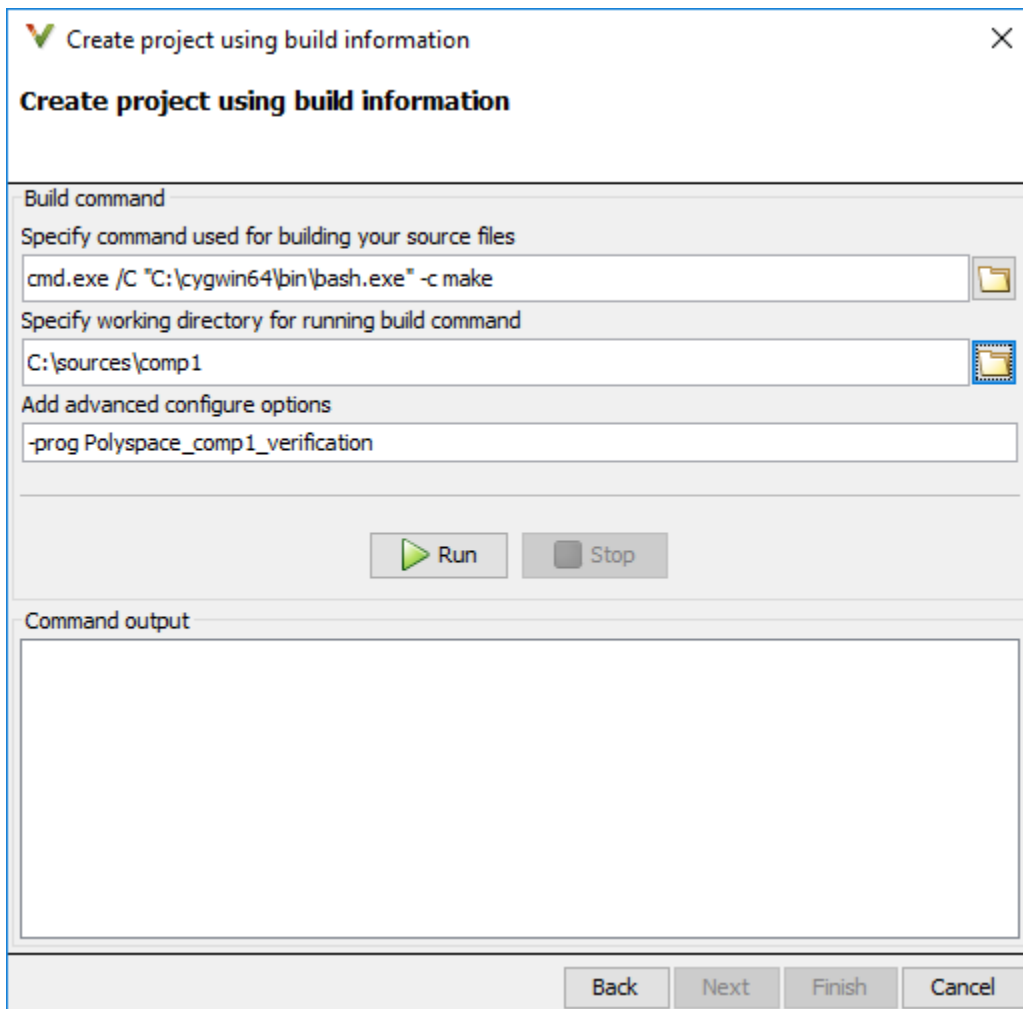
Because source file paths are recomputed relative to a project path, you can commit a Polyspace project to a version control system along with your source files. When you check out the project from your version control system and open a local copy of the project, all source file paths are recomputed based on the new location of the project. The project now points to a local copy of the source files.

Add Sources from Build Command

Select **File > New Project**. Select **Create from build command**.

After providing a project name and location, on the next window, enter this information:

- The build command, exactly as you run it on your code.
- The folder from which you run your build command.



When you click **Run**, Polyspace runs the build command and extracts the information for creating a Polyspace project, specifically, source paths and compiler information.

If you build your source code within an IDE such as Visual Studio®, in the field for specifying the build command, enter the path to your executable, for instance, C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\VCExpress.exe. When you click **Run**, Polyspace opens your IDE. In your IDE, perform a complete build of your code. When you close your IDE, Polyspace extracts your source paths and compiler information. See also “Create Polyspace Projects from Visual Studio Build” on page 2-9.

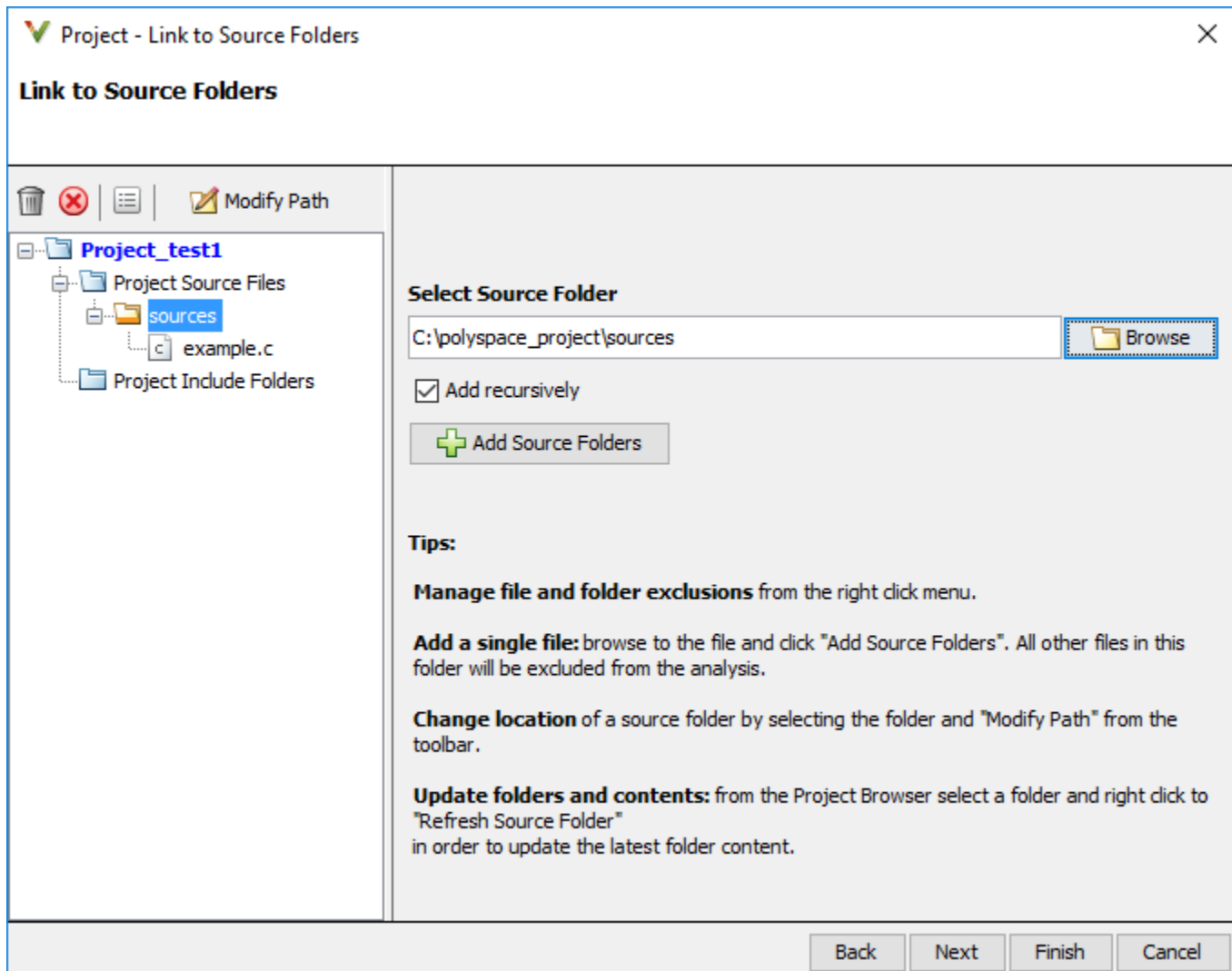
When you create a project from your build command, the **Project Browser** pane displays your source folders but not the include folders. In case you want to verify that your include folders were extracted, open the project file (with extension .psprj) in a text editor.

You can use additional options to modify the default project creation from build command. For instance, to create a Polyspace project despite build errors, in the **Add advanced configure options** field, enter the option `-allow-build-error`. To look up allowed options, see `polyspace-configure`.

Add Sources Manually

Select **File > New Project**.

After providing a project name and location, on the next window, enter or navigate to the root folder containing your source files. After selecting the **Add recursively** box, click **Add Source Folders**. All files in the folder and subfolders are added to your project. To exclude specific files or folders from analysis, right-click the files or folders and select **Exclude Files**.



On the next window, add include folders. The analysis looks for include files relative to the include folder paths that you specify. For instance, if your code contains the preprocessor directive `#include<../mylib.h>` and you include the folder:

```
C:\My_Project\MySourceFiles\Includes
```

the folder `C:\My_Project\MySourceFiles` must contain a file `mylib.h`.

For Standard Library headers such as `stdio.h`, if you know the path to the headers from your compiler, specify them explicitly. Otherwise, the analysis uses Polyspace implementation of the

Standard Library headers, which in some special cases, might not match your compiler implementation. See also “Provide Standard Library Headers for Polyspace Analysis” on page 13-20.

Your project file with source and include folders are displayed in the **Project Browser** pane. Later, if you add files to one of these folders, you can update your project. Right-click the folder that you want to update, or the entire **Project Source Files** folder, and select **Refresh Source Folder**.

You can also right-click to exclude files or add more folders to the project. The files that you add the first time are copied to the first module in your project. If you add new files later, you must explicitly right-click them and add them to a module.

Add Source Files Based on AUTOSAR Design Specifications

If your code implements AUTOSAR software components, you can provide the top level folder containing your AUTOSAR design specifications and folders containing the source code implementation of those specifications.

- 1 Select **File > New**. In the Project-Properties window, select **Create from AUTOSAR specification**.
- 2 Specify the top level folder containing your ARXML files and all the folders containing source files.

For details, see “Run Polyspace on AUTOSAR Code” (Polyspace Code Prover).

See Also

More About

- “Run Analysis in Polyspace Desktop User Interface” on page 3-2
- “Create Polyspace Projects from Visual Studio Build” on page 2-9
- “Provide Standard Library Headers for Polyspace Analysis” on page 13-20

Contents of Polyspace Project and Results Folders

This topic applies only to the Polyspace desktop products..

A Polyspace analysis generates files that contain information about configuration options and analysis results.

If you run the analysis from the Polyspace user interface, you can group results into modules in a single project. The project, module and results can correspond to physical folder locations. If you run the analysis from the command line, you can only specify the path to a results folder (using the option `-results-dir`). You have to group related results using appropriate conventions for creating folders.

File Organization

The organization of Polyspace files in the physical folder location follows the hierarchy displayed in the Polyspace user interface: project > module > results. The project folder contains a subfolder for each module. In each module folder, there is one or more result subfolder, named `Result_#`.

The number of result folders depends on whether you overwrite or retain previous results for each new run. To use a different folder naming convention or different storage location for results, select **Tools > Preferences** and use the options on the **Project and Results Folder** tab.

The project folder has the project file with extension `.psprj`. If you open a project from a previous release in the user interface, the project is upgraded for the new release. A backup of the old project file is saved with the extension `.bak.psprj`.

Files in the Results Folder

Some of the files and folders in the results folder are described below. The contents of the results folder are the same irrespective of whether you run the analysis from the user interface or command line.

- `Polyspace_release_project_name_date-time.log` — A log file associated with each analysis.
- `ps_results.psbf` — An encrypted file containing your Polyspace results. Open this file in the Polyspace environment to view your results.
- `ps_sources.db` — A non-encrypted database file listing source files and macros.
- `drs-template.xml` — A template generated when you use constraint specification.
- `ps_comments.db` — An encrypted database file containing your comments and justifications.
- `comments_bak` — A subfolder used to import comments between results.
- `.status` and `.settings` — Two folders that store files required to relaunch the analysis.
- `Polyspace-Doc` — When you generate a report, by default, your report is saved in this folder with the name `ProjectName_ReportType`. For example, a developer report in PDF format would be, `myProject_Developer.pdf`.

Note that by default, the results folder is cleaned up and repopulated at each run. To avoid accidental removal of files during the cleanup, instead of using an existing folder that contains other files, specify a dedicated folder for the Polyspace results.

See Also

-results-dir

Create Polyspace Projects from Visual Studio Build

In this section...

“Create Polyspace Project from Build in Visual Studio Developer Command Prompt” on page 2-9

“Create Polyspace Project from Build in Visual Studio IDE” on page 2-10

This topic shows how to create a Polyspace project for use with the Polyspace desktop products. If using the Polyspace as You Code plugin for single-file analysis in Visual Studio, see “Run Polyspace as You Code in Visual Studio and Review Results”.

If you develop in the Visual Studio IDE, you can trace the commands running underneath your Visual Studio build and create a Polyspace project. This method of creating a project automatically adds source files and compilation options from the Visual Studio project to the Polyspace project.

Note that to accurately reflect your Visual Studio project, you must run a complete build of your project and not an incremental build. An incremental build only rebuilds sources that changed since the previous build and might lead to incomplete Polyspace projects.

You can create a Polyspace project by tracing a Visual Studio build at the command line or within an IDE. Although the latter approach might be simpler, building within an IDE introduces additional complications when tracing the build. Therefore, calling the build command directly at the command line is the recommended approach.

Create Polyspace Project from Build in Visual Studio Developer Command Prompt

To create a Polyspace project, you simply have to prepend `polyspace-configure` to your regular build command. For instance, suppose you have a Visual Studio project `TestProject.vcxproj`. To create a Polyspace project:

- 1 Open the Visual Studio developer command prompt. For instance, in Windows®, start typing Developer Command Prompt for VS 2017.

This command prompt is similar to a regular command prompt but with all Visual Studio environment variables appropriately set up.

- 2 Perform a full build of your Visual Studio project: at the command prompt:

```
msbuild TestProject.vcxproj /t:Rebuild
```

This step is optional. Ensuring that the build completes successfully by itself allows you to create a Polyspace project from an error-free build.

- 3 Run the complete build command from the previous step but prepended with the `polyspace-configure` command:

```
polyspace-configure msbuild TestProject.vcxproj /t:Rebuild
```

For the above command to work, add the path `polyspaceroot\polyspace\bin` to the Path environment variable in Windows. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2023a`.

Instead of a project, you can also run `polyspace-configure` on the full build of a solution. However, a solution consists of multiple projects, each of which might generate a separate

executable. In this situation, `polyspace-configure` generates a project that mixes source files contributing to separate executables. To avoid the issue:

- If all projects in the solution generate a single process, for instance, when the solution generates an executable for a GUI app and a DLL containing the engine for the app, you can run `polyspace-configure` on the full build of the solution. In all other cases, run `polyspace-configure` on specific projects in the solution.

For instance, if a solution `ExampleProject` contains two projects `AProject` and `AnotherProject`, you can run `polyspace-configure` from the folder containing the solution as follows:

```
polyspace-configure -prog AProject ^
    msbuild ExampleProject/AProject.vcxproj /t:Rebuild
polyspace-configure -prog AnotherProject ^
    msbuild ExampleProject/AnotherProject.vcxproj /t:Rebuild
```

These commands generate two Polyspace projects, `AProject.psprj` and `AnotherProject.psprj`.

- Instead of creating a Polyspace project to run analysis, you can run the analysis using options files. See also “Options Files for Polyspace Analysis” on page 12-5. If you take the options file approach to run Polyspace, you can first run `polyspace-configure` on a Visual Studio solution to generate one options file per project in the solution.

For instance, if a solution `ExampleProject` contains two projects `AProject` and `AnotherProject`, you can run `polyspace-configure` as follows:

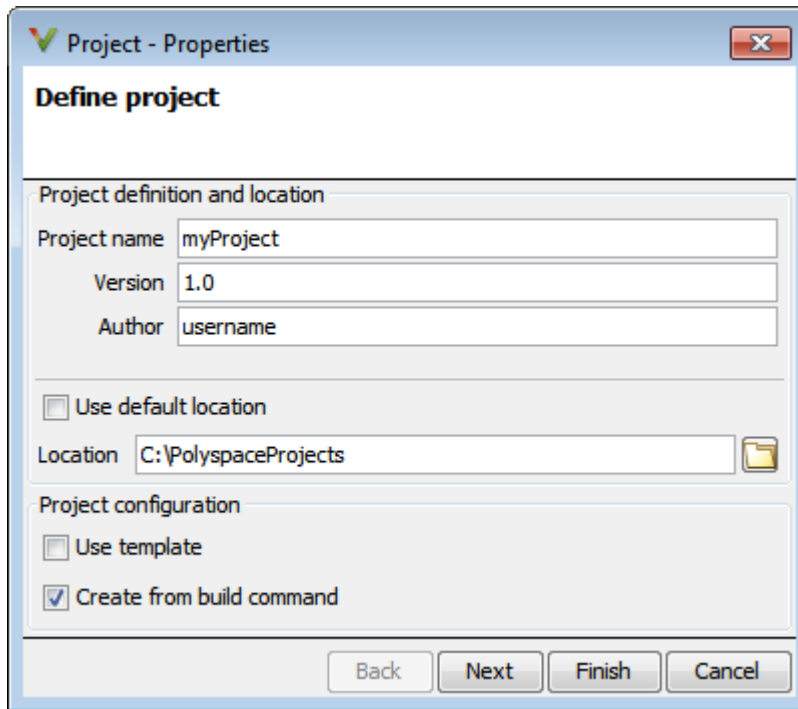
```
polyspace-configure -module -output-options-path . ^
    msbuild ExampleProject.sln /t:Rebuild
```


This command generates two options files, `AProject_exe.psopts` and `AnotherProject_exe.psopts`. You can continue the analysis using these options files.

Create Polyspace Project from Build in Visual Studio IDE

To create a Polyspace project, you can also open the Visual Studio IDE from within Polyspace and perform a full build within the IDE.

- 1 In the Polyspace interface, select **File > New Project**.
- 2 In the Project - Properties window, under **Project Configuration**, select **Create from build command** and click **Next**.

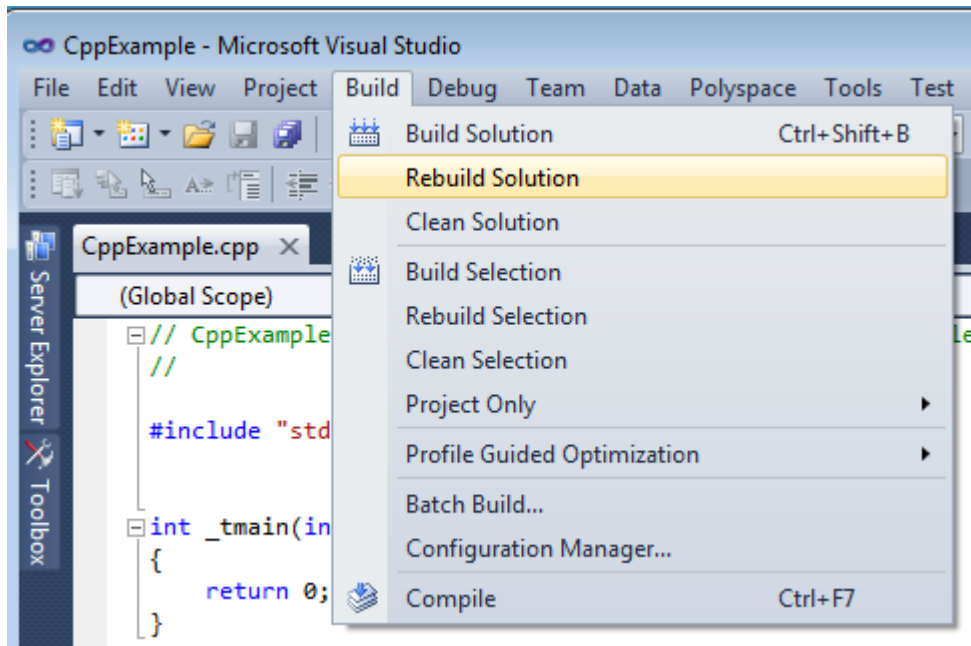


- 3 In the field **Specify command used for building your source files**, enter the full path to the Visual Studio executable. For instance, "C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\devenv.exe".
- 4 In the field **Specify working directory for running build command**, enter a folder to which you have write access, for instance, C:\temp\Polyspace. Click .

This action opens the Visual Studio environment.

- 5 In the Visual Studio environment, create and build a Visual Studio project.

If you already have a Visual Studio project, open the existing project and build a clean solution. For instance, to build a clean solution in Visual Studio 2012, select **BUILD > Rebuild Solution**.



- 6 After the project builds, close Visual Studio.

Polyspace traces your Visual Studio build and creates a Polyspace project.

The Polyspace project contains the source files from your Visual Studio build and the relevant **Target & Compiler** options.

- 7 If you update your Visual Studio project, to update the corresponding Polyspace project, on the **Project Browser**, right-click the project name and select **Update Project**.

See Also

polyspace-configure

More About

- “Troubleshoot Project Creation from Visual Studio Build” on page 32-20

Create Project in Polyspace Desktop User Interface Using Configuration Template

This topic shows how to export and reuse a configuration in the user interface of the Polyspace desktop products.

- *If using the Polyspace Server products, see “Set Up Bug Finder Analysis on Servers During Continuous Integration”.*
- *If using Polyspace as You Code, see “Set Up Polyspace Analysis in IDEs”.*

A configuration template is a predefined set of analysis options for a specific compilation environment.

Why Use Templates

Use templates to simplify your project setup. For instance, after you configure a project for a specific compilation environment, you can create a template out of the configuration. Using the template, you can reuse the configuration for projects that have the same compilation environment.

When creating a new project, you can do one of the following:

- Use an existing template to automatically set analysis options for your compiler.

Polyspace software provides predefined templates for common compilers such as IAR, Kiel, Visual and VxWorks. For additional templates, see Polyspace Compiler Templates.
- Set analysis options manually. You can then save your options as a template and reuse them later. You can also share the template with other users and enforce consistent usage of Polyspace Bug Finder in your organization.

Use Predefined Template

- 1 Select **File > New Project**.
- 2 On the Project - Properties dialog box, after specifying the project name and location, under **Project configuration**, select **Use template**.
- 3 On the next screen, select the template that corresponds to your compiler. For further details on a template, select the template and view the **Description** column on the right.

If your compiler does not appear in the list of predefined templates, select **Baseline_C** or **Baseline_C++**.

- 4 On the next screen, add your source files and include folders.

Create Your Own Template

This example shows how to save a configuration from an existing project and create a new project using the saved configuration.

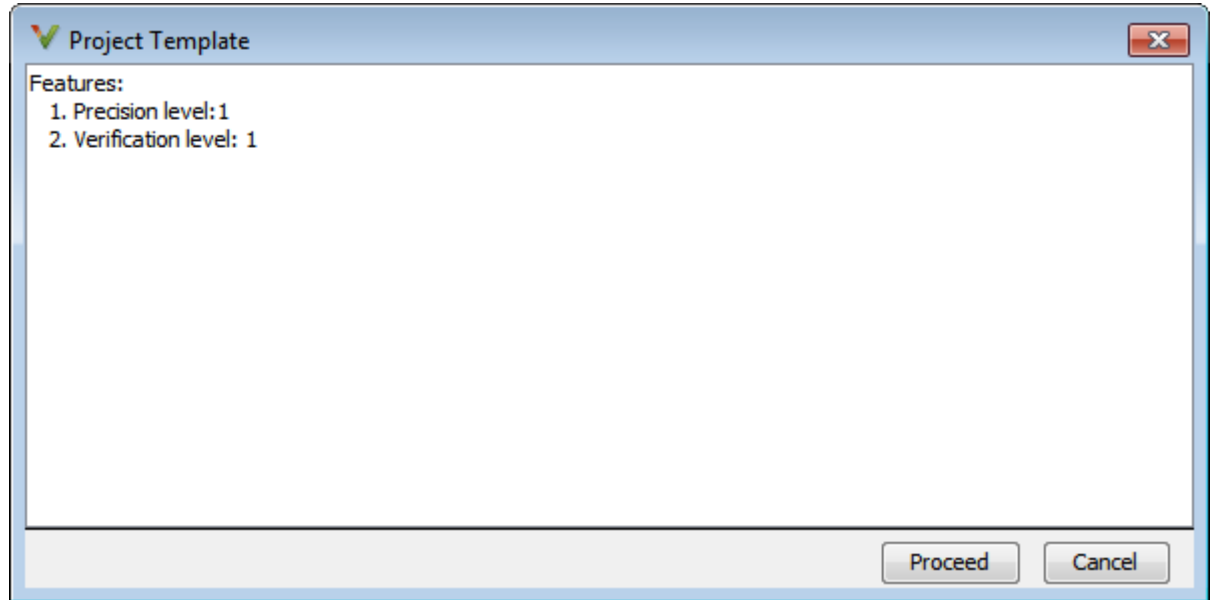
- To create a template from a project that is open on the **Project Browser** pane:
 - 1 Right-click the project configuration that you want to use, and then select **Save As Template**.

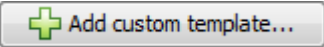
- 2 Enter a description for the template, then click **Proceed**. Save your template file.

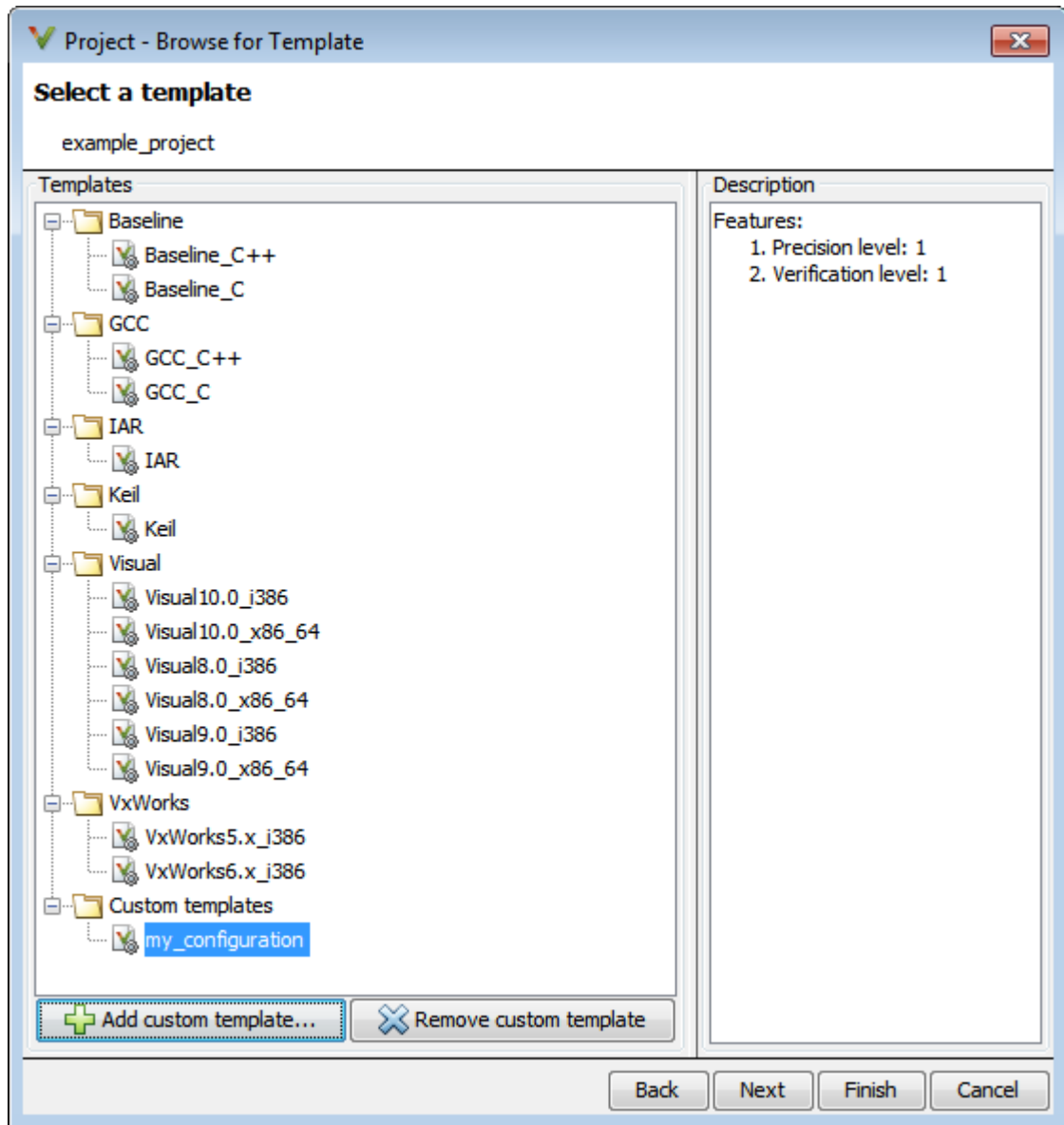
Suppose you create a Code Prover configuration template that runs Code Prover analysis to a precision level of 1 and a verification level of 1. See:

- Precision level (-00 | -01 | -02 | -03) (Polyspace Code Prover)
- Verification level (-to) (Polyspace Code Prover)

You can enter this description for the template.



- When you create a new project, to use a saved template:
 - 1 Select . The button is rectangular with a green plus sign icon and the text "Add custom template...".
 - 2 Navigate to the template that you saved earlier, and then click **Open**. The new template appears in the **Custom templates** folder on the **Templates** browser. Select the template for use.



Sharing Project Templates

A configuration template stores all options set on the **Configuration** pane in the Polyspace desktop user interface. If you share the template, another user who uses the template can benefit from those options.

Note however that options that refer to specific files point to their absolute paths. If a shared template sets one of those options, the corresponding file must also be shared. Preferably, the shared file must be in the same location as when the template was created, otherwise end-users have to modify the template to point to a new location. If you set one of those options in a configuration template that is meant to be shared with other users, make sure that the corresponding file is in a location accessible to the end-users. Some common options that refer to specific files are:

- Command/script to apply to preprocessed files (`-post-preprocessing-command`) and Command/script to apply after the end of the code verification (`-post-analysis-command`)
- Set checkers by file (`-checkers-selection-file`) and `-checkers-activation-file`
- Constraint setup (`-data-range-specifications`)
- Command-line-only options such as `-options-file` and `-code-behavior-specifications`. In the Polyspace user interface, you enter these options in the `Other` field.

See Also

More About

- “Specify Polyspace Analysis Options” on page 12-2
- “Complete List of Polyspace Bug Finder Analysis Engine Options”

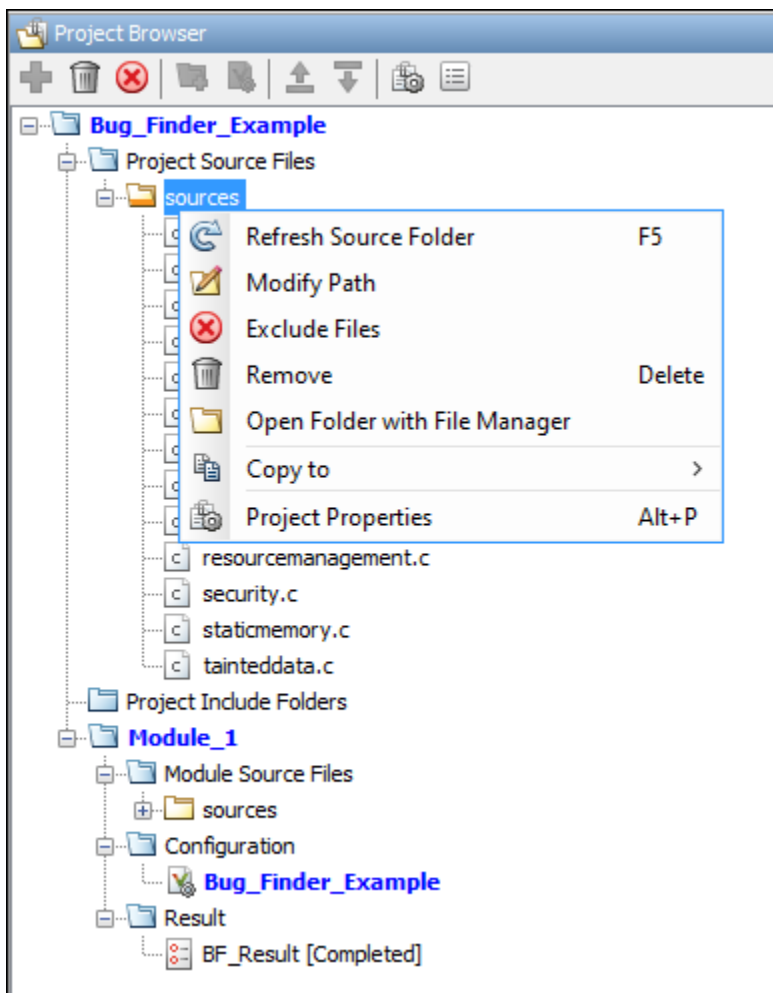
Update Project in Polyspace Desktop User Interface

This topic shows how to update a project in the user interface of the Polyspace desktop products.

- If using the Polyspace Server products, see “Set Up Bug Finder Analysis on Servers During Continuous Integration”.
- If using Polyspace as You Code, see “Set Up Polyspace Analysis in IDEs”.


To analyze your C/C++ source files with Bug Finder or Code Prover in the Polyspace user interface, you create a Polyspace project. During development, you can simply update this project and rerun the analysis for updated results. This topic describes the updates that you can make.

To begin updates, right-click your project on the **Project Browser** pane. You see a different set of options depending on the node that you right-click.



Change Folder Path

If you have moved the source folder that you added to your project, modify the path in your Polyspace project. You can also modify the folder path to point to a different version of the code in your version control system.

In the **Project Browser**, right-click the top sources folder  and select **Modify Path**. Change the path to the new location.

To resync the files under this source folder, right-click your source folder and select **Refresh Source Folder**.

Refresh Source List

If you made changes to files in a folder already added to the project, you do not need to re-add the folder to your project. Refreshing your source file list looks for new files, removed files, and moved files.


Right-click your source folder and select **Refresh Source Folder**. The files in your Polyspace project refresh to match your file system.

Refresh Project Created from Build Command

If you created your project automatically from your build system, to update the project later by rerunning your build command, right-click the project folder and select **Update Project**.

You see the information that you entered when creating the original project. Click **Run** to retrace your build command and recreate the Polyspace project.

Add Source and Include Folders

If you want to change which files or folders are active in your project without removing them from your project tree, right-click the file or folder and select **Exclude Files**. The file appears with an  symbol in your project indicating it is not considered for analysis. You can reinclude the files for analysis by right-clicking and selecting **Include Files**.

If you want to add additional source folders or include folders, right-click your project or the **Source** or **Include** folder in your project. Select **Add Source Folder** or **Add Include Folder**.

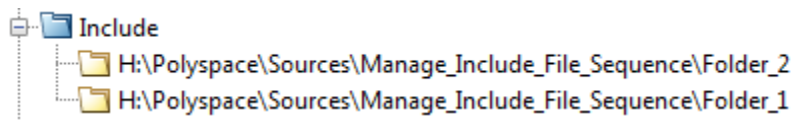
Before running an analysis, you must copy the source files to a module. Select the source files that you want to copy. To select multiple files together, press the **Ctrl** key while selecting the files. Right-click your selection. Select **Copy to > Module *n***. *n* is the module number.



Manage Include File Sequence

You can change the order of include folders to manage the sequence in which include files are compiled.

When multiple include files by the same name exist in different folders, you might want to change the order of include folders instead of reorganizing the contents of your folders. For a particular include file name, the software includes the file in the first include folder under **Project_Name > Include**.

In the following figure, Folder_1 and Folder_2 contain the same include file `include.h`. If your source code includes this header file, during compilation, Folder_2/`include.h` is included in preference to Folder_1/`include.h`.



To change the order of include folders, in your project, expand the **Include** folder. Select the include folder or folders that you want to move. To move the folder, click either  or .

See Also

Related Examples

- “Add Source Files for Analysis in Polyspace Desktop User Interface” on page 2-2

Organize Layout of Polyspace Desktop User Interface

The Polyspace user interface has two default layouts of panes.

The default layout for project setup has the following arrangement of panes:

Project Browser	Configuration
	Output Summary

The default layout for results review has the following arrangement of panes:

Results List	Result Details
	Dashboard

You can create and save your own layout of panes. If the current layout of the user interface does not meet your requirements, you can use a saved layout.

You can also change to one of the default layouts of the Polyspace user interface. Select **Window > Reset Layout > Project Setup** or **Window > Reset Layout > Results Review**.

Create Your Own Layout

To create your own layout, you can close some of the panes, open some panes that are not visible by default, and move existing panes to new locations.

To open a closed pane, select **Window > Show/Hide View > pane_name**.

To move a pane to another location:


1 Float the pane in one of three ways:

- Click and drag the blue bar on the top of the pane to float all tabs in that pane.


For instance, if **Project Browser** and **Results List** are tabbed on the same pane, this action floats the pane together with its tabs.

- Click and drag the tab at the bottom of the pane to float only that tab.

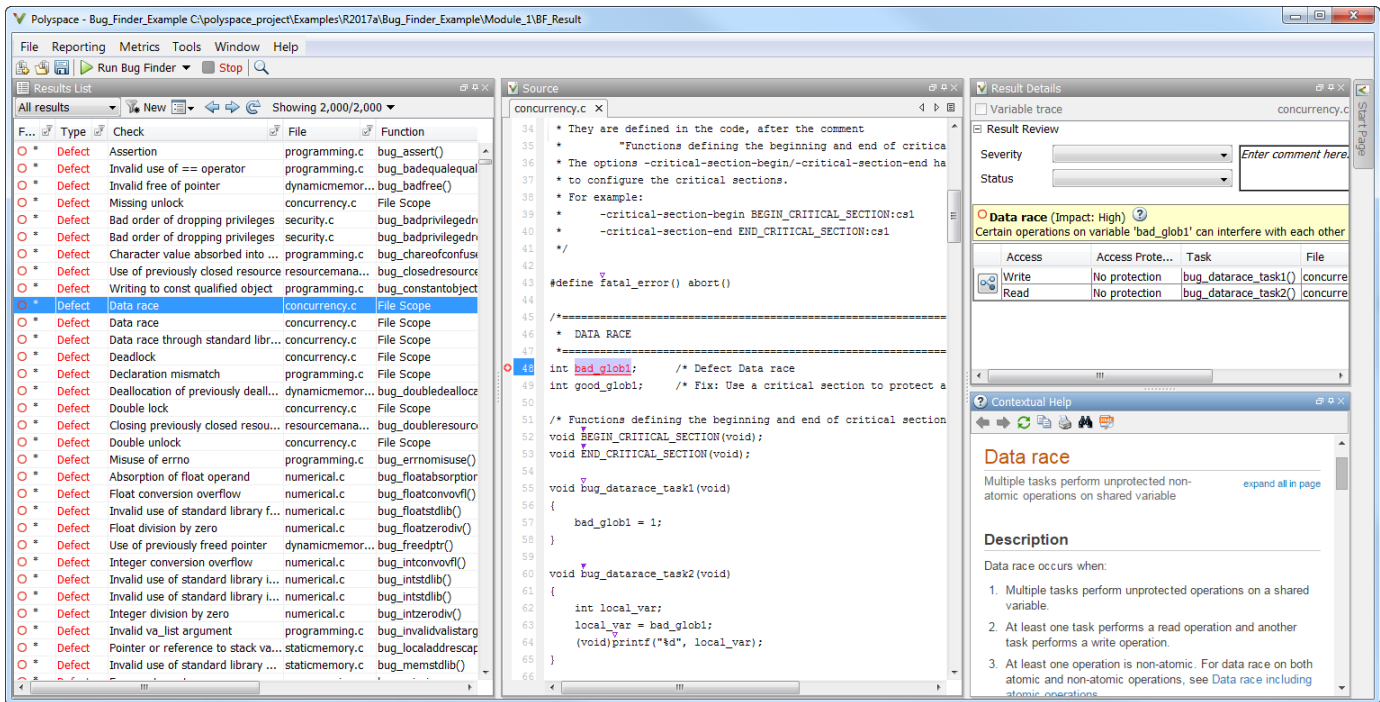
For instance, if **Project Browser** and **Results List** are tabbed on the same pane, dragging out **Project Browser** creates a pane with only **Project Browser** on it and floats this new pane.

- Click  on the top right of the pane to float all tabs in that pane.

2 Drag the pane to another location until it snaps into a new position.

If you want to place the pane in its original location, click  in the upper-right corner of the floating pane.

For instance, you can create your own layout for reviewing results.



Save and Reset Layout

After you have created your own layout, you can save it. You can change from another layout to this saved layout.

- To save your layout, select **Window > Save Current Layout As**. Enter a name for this layout.
- To use a saved layout, select **Window > Reset Layout > *layout_name***.
- To remove a saved layout from the **Reset Layout** list, select **Window > Remove Custom Layout > *layout_name***.

See Also

More About

- “Customize Polyspace Desktop User Interface” on page 2-22

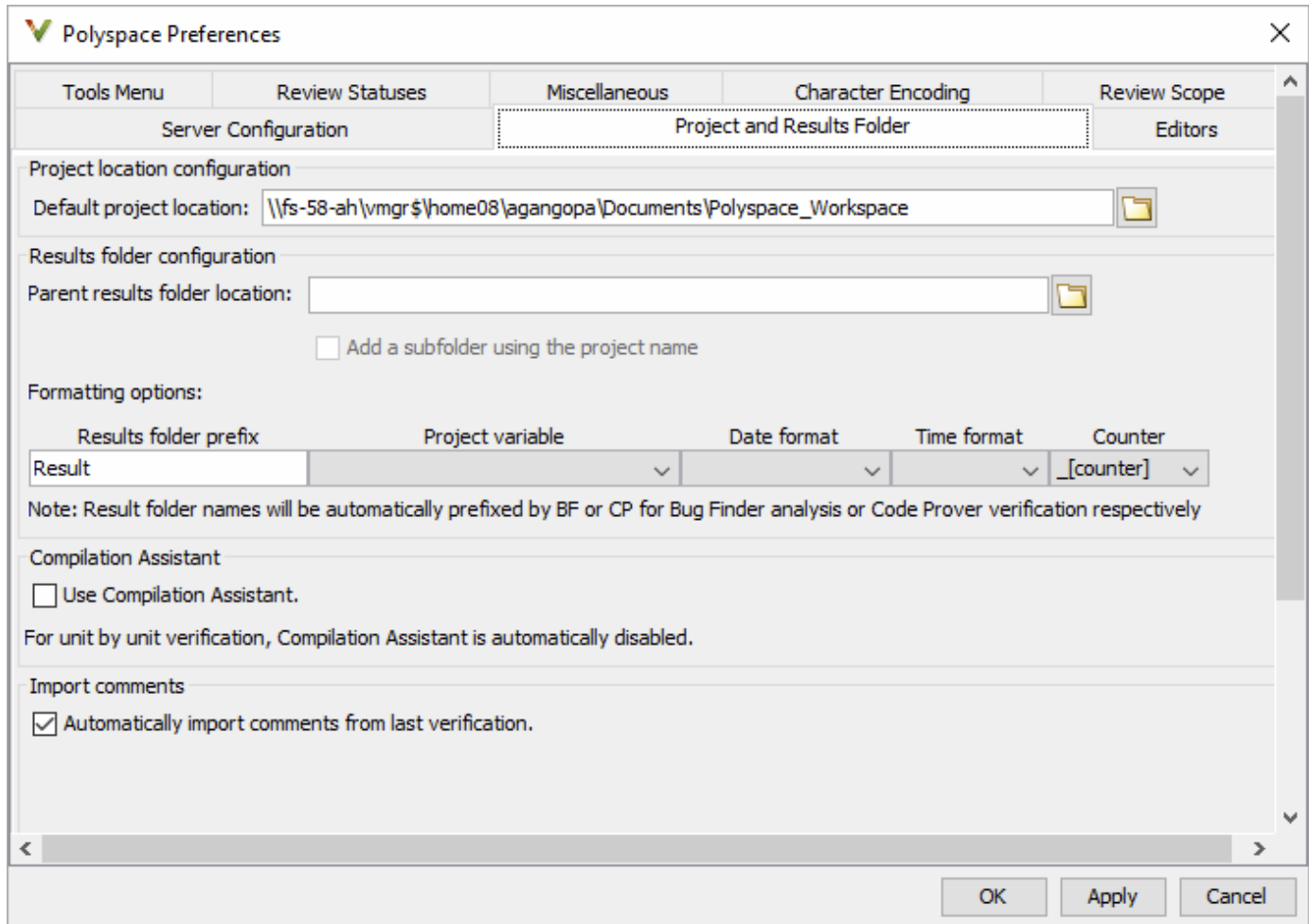
Customize Polyspace Desktop User Interface

In this section...

“Possible Customizations” on page 2-22

“Storage of Polyspace User Interface Customizations” on page 2-24

You can customize various aspects of the Polyspace user interface, for instance, default project storage locations or default font size of source code. Select **Tools > Preferences**.



Possible Customizations

Change Default Font Size

To change the default font size in the Polyspace user interface, select the **Miscellaneous** tab.

- To increase the font size of labels on the user interface, select a value for **GUI font size**.

For example, to increase the default size by 1 point, select +1.

- To increase the font size of the code on the **Source** pane and the **Code Editor** pane, select a value for **Source code font size**.

When you restart Polyspace, you see the increased font size.

Specify External Text Editor

You can change the default text editor for opening source files from the Polyspace interface. By default, if you open your source file from the user interface, it opens on a **Code Editor** tab. If you prefer editing your source files in an external editor, you can change this default behavior.

To change the text editor, select the **Editors** tab. From the **Text editor** drop-down list, select **External**. In the **Text editor** field, specify the path to your text editor. For example:

```
C:\Program Files\Windows NT\Accessories\wordpad.exe
```

To make sure that your source code opens at the correct line and column in your text editor, specify command-line arguments for the editor using Polyspace macros, \$FILE, \$LINE and \$COLUMN. Once you specify the arguments, when you right-click a check on the **Results List** pane and select **Open Editor**, your source code opens at the location of the check.

Polyspace has already specified the command-line arguments for these editors: Emacs, Notepad++ (Windows only), UltraEdit, VisualStudio, WordPad (Windows only) or gVim. If you are using one of these editors, select it from the **Arguments** drop-down list. If you are using another text editor, select **Custom** from the drop-down list, and enter the command-line options in the field provided.

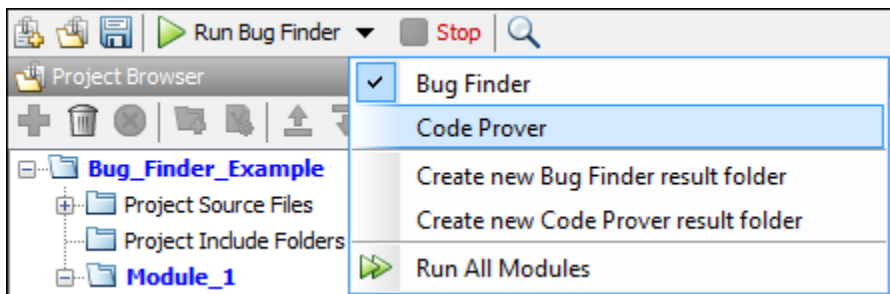
For console-based text editors, you must create a terminal. For example, to specify **vi**:

- 1 In the **Text Editor** field, enter `/usr/bin/xterm`.
- 2 From the **Arguments** drop-down list, select **Custom**.
- 3 In the field to the right, enter `-e /usr/bin/vi $FILE`.

To revert back to the built-in editor, on the **Editors** tab, from the **Text editor** drop-down list, select **Built In**.

Create Naming Convention for Results Folder

By default, results are stored in a subfolder of the project folder. When you run an analysis, you can overwrite the results of the previous run or create a new results folder.



You can customize the results folder on the **Project and Results Folder** tab in these ways:

- If you create a new results folder for each run, you can define a naming convention for the folder. To specify a results folder naming convention, in the section **Results folder configuration**, use the options under **Formatting options** to create a naming convention for results folders.

For instance, the results folder naming convention below uses the module name and date and time of analysis. So, a Bug Finder result folder using this convention has a name such as BF_Result_module_2_01_01_2020_22_30.

Results folder prefix	Project variable	Date format	Time format	Counter
Result	_[module]	_[dd_MM_yyyy]	_[HH_mm]	

Note: Result folder names will be automatically prefixed by BF or CP for Bug Finder analysis or Code Prover verification respectively

- You can store results separately from projects. In the section **Results folder configuration**, you can specify a root folder for storing results and store per-project results in subfolders of the root folder:
 - Specify the root folder for **Parent results folder location**.
 - Select the option **Add a subfolder using the project name**.

Create Custom Review Status

When reviewing Polyspace results, you can assign a status such as **To fix** or **Justified**. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 22-2.

You can also create and assign custom statuses. To create a new status:

- Select the **Review Statuses** tab.
- Enter the status in the **Add a new status** field and click **Add**.

Optionally, to specify that Polyspace should consider results with this review status justified, select the checkbox next to the **Add** button. See also “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 22-2.

Storage of Polyspace User Interface Customizations

The software stores the settings that you specify through the Polyspace Preferences in the following file:

- Windows: `$Drive\Users\%User\AppData\Roaming\MathWorks\MATLAB\%Release\Polyspace\polyspace.prf`
- Linux®: `/home/%User/.matlab/%Release/Polyspace/polyspace.prf`

Here, `$Drive` is the drive where the operating system files are located such as `C:`, `%User` is the username and `%Release` is the release number.

The following file stores the location of all installed Polyspace products across various releases:

- Windows: `$Drive\Users\%User\AppData\Roaming\MathWorks\MATLAB\polyspace_shared\polyspace_products.prf`
- Linux: `/home/%User/.matlab/polyspace_shared/polyspace_products.prf`

Upload Results to Polyspace Access

Polyspace Access offers a centralized database where you can store Polyspace analysis results for sharing and collaborative reviews. After you upload results, open the Polyspace Access user interface to view statistics about the quality of your code and to triage and review individual results.

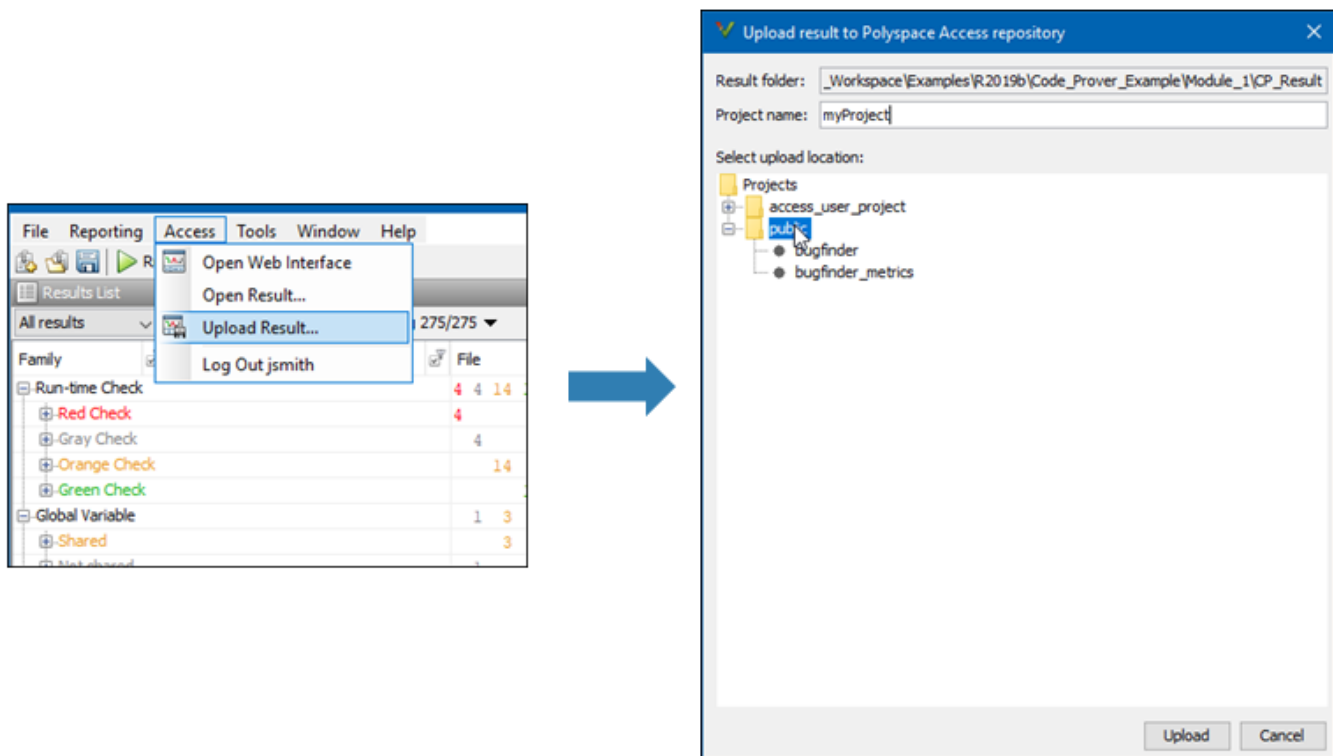
Polyspace assigns a unique run ID to each analysis run that you upload and increments the run ID with each upload to any project. If you use an automation tool such as Jenkins to upload results, the Polyspace Access run ID is not related to the tool job ID.

Note You can upload up to 2GB of results per upload to Polyspace Access.

Upload Results from Polyspace Desktop Client

Before you upload results, you must configure the Polyspace desktop client to communicate with Polyspace Access. See “Register Polyspace Desktop User Interface”.

To upload analysis results to the Polyspace Access database from the Polyspace desktop client, select a set of results in the **Project Browser** pane or open the results in the **Results List** pane. Go to **Access > Upload Results** and follow the prompts. If you get a login request, use your Polyspace Access login credentials.



You can also upload results to Polyspace Access by selecting a result in the **Project Browser** pane and using the context menu.

After you upload results to Polyspace Access, if you open a local copy of the results in the desktop interface, you cannot make changes to the **Status**, **Severity**, or comment fields. To make changes to the **Status**, **Severity**, or comment fields, open the results from Polyspace Access by going to **Access > Open Results**.

Once you save the changes you make to these fields in the desktop interface, the changes are reflected in the Polyspace Access web interface. To create custom statuses, see “Add Custom Status in Polyspace Access Project” on page 26-3.

Upload Results at Command Line

You can upload results from the command line only if they are generated with Polyspace Bug Finder Server or Polyspace Code Prover™ Server.

To upload analysis results to Polyspace Access from the DOS or UNIX command line, use the `polyspace-access` binary. See `polyspace-access`.

In the command, specify the path of the folder under which the `.psbf`, `.pscp`, or `.rte` results file is stored. For instance, to upload Polyspace Bug Finder results stored in the file `BF_results\ps_results.psbf`, use this command:

```
polyspace-access %login% -upload BF_results\ps_results.psbf
```

The command uploads the results to the **public** folder of the Polyspace Access database and outputs information about the upload including an `ACCESS URL`. You can use the URL to view the uploaded results in the Polyspace Access interface. To upload results to a different folder, use the `-parent-project` option.

Here, `%login%` is a variable that stores the login credentials and other connection information. To configure this variable, see “Encrypt Password and Store Login Options in a Variable”.

For faster uploads, store your analysis results in a dedicated results folder by using option `-results-dir` when you run the analysis. If you store results in a folder that contains a large number of files unrelated to Polyspace analysis results, for example the root folder of your repository, Polyspace Access takes longer to upload the results.

Results Upload Compatibility and Permissions

Results Compatibility

You cannot upload analysis results to a Polyspace Access version that is older than the version of the Polyspace product that generated the results. For instance, you cannot upload results generated with a Polyspace product version R2019b to a Polyspace Access version R2019a.

If you upload results generated with a Polyspace product version R2018b or earlier, you cannot view these results in the Polyspace Access **REVIEW** perspective. To review R2018b or earlier results that you uploaded to Polyspace Access, see “Open Polyspace Access Results in a Desktop Interface” on page 28-2.

You can upload results to an existing Polyspace Access project only if those results were generated by the same type of analysis. For instance, you cannot upload results of a Bug Finder analysis to a project that contains Code Prover results.

User Permissions for Uploaded Results

You are the project **Owner** for all the results that you upload. The project **Owner** or an **Administrator** must add other users as **Contributor** to grant them permission to see those results, unless you upload the results to a folder that other users already have permission to see.

Results that you upload to the **public** folder are visible to all Polyspace Access users. For more information, see “Manage Project Permissions” on page 27-3.

See Also

polyspace-access

More About

- “Register Polyspace Desktop User Interface”

Run Polyspace Analysis on Desktop

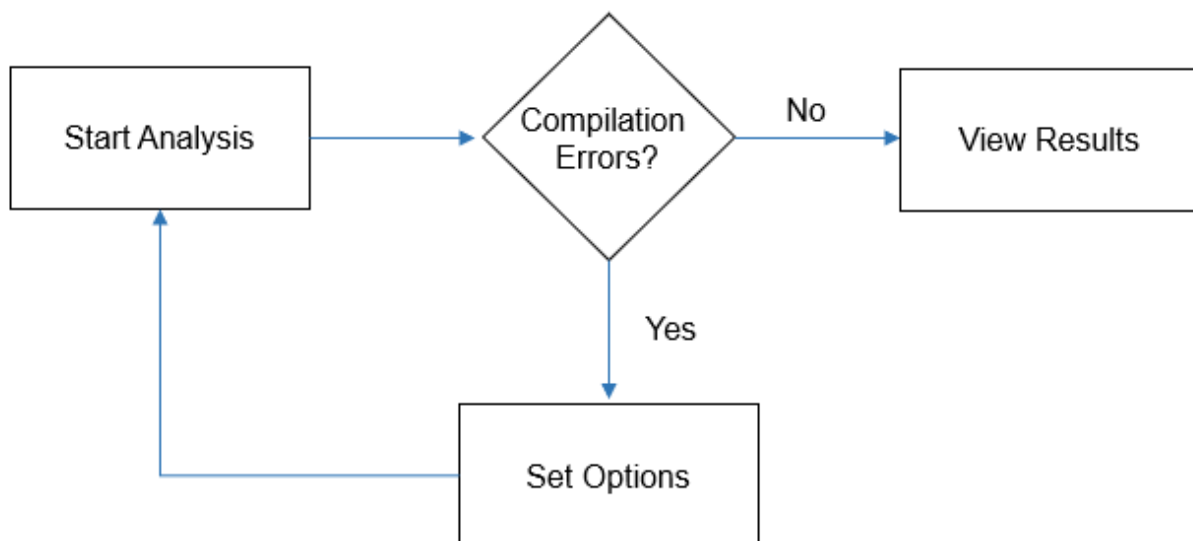
Run Analysis in Polyspace Desktop User Interface

This topic shows how to run an analysis in the user interface of the Polyspace desktop products.

- If using the Polyspace Server products, see “Set Up Bug Finder Analysis on Servers During Continuous Integration”.
- If using Polyspace as You Code, see “Set Up Polyspace Analysis in IDEs”.

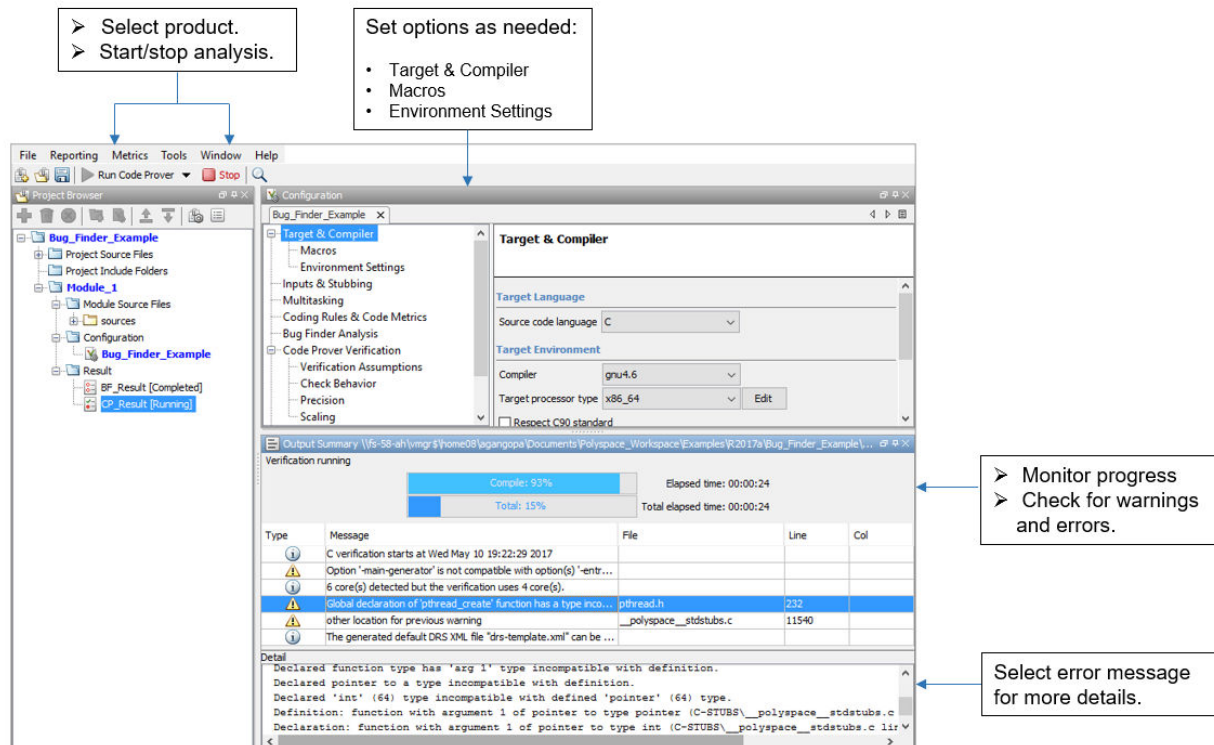
This topic describes how to run an analysis in the Polyspace user interface, monitor progress, fix compilation issues, and open analysis results as available.

After you specify your source files and compiler on page 2-2, start the Polyspace analysis. During analysis, Polyspace first compiles your code, and then checks for bugs (Bug Finder) or proves code correctness (Code Prover). If you encounter compilation errors, read the error message and diagnose the root cause of the error. To resolve the errors, you often have to set some Polyspace configuration options and rerun the analysis.



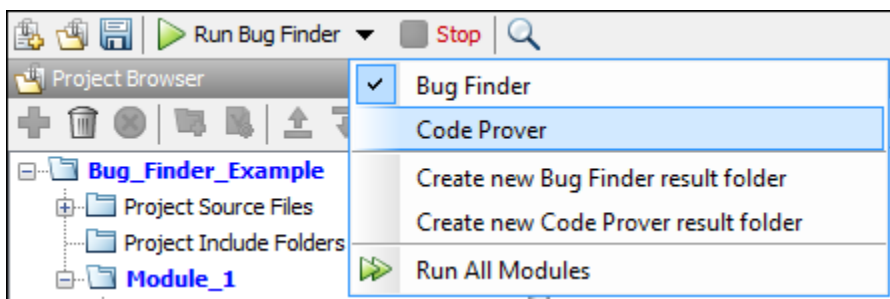
Arrange Layout of Windows for Project Setup

To set up a convenient distribution of windows, in the Polyspace user interface, select **Window > Reset Layout > Project Setup**.



Set Product and Result Location

To switch products or create a separate folder for each run, select options from the drop-down list beside the **Run** button. For instance, to avoid overwriting previous results each time that you run Bug Finder and keep existing results, select **Create new Bug Finder result folder**.



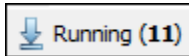
The results are stored in subfolders `Module_1`, `Module_2`, and so on in the project folder. To find the physical location of the project folder, right-click a project on the **Project Browser** pane and select **Open Folder with File Manager**.

To use a different folder naming convention or a different storage location for results, select **Tools > Preferences** and use the options on the **Project and Results Folder** tab. See also “Create Naming Convention for Results Folder” on page 2-23.

Start and Monitor Analysis

If your project has multiple modules, select the module that you want to analyze. To start the analysis, select **Run Bug Finder** or **Run Code Prover**. Monitor progress on the **Output Summary** pane.


- Bug Finder: You can see some results after partial analysis because certain defect checkers do not need cross-functional information and can show results as soon as a function is analyzed. If results are available while the analysis is still running, you see this icon beside the **Run Bug Finder** button:



The icon indicates the number of results available. To open the results, click the icon. Once the analysis is over, the **Running** label in the icon changes to **Completed**. To reload the full set of results, click the icon again.

- Code Prover: You can see results only after the analysis is complete. Code Prover is more likely to report compilation errors because it does a more rigorous analysis and must follow stricter rules for compilation. The progress bar distinguishes between the various phases of analysis starting from compilation.

Fix Compilation Errors

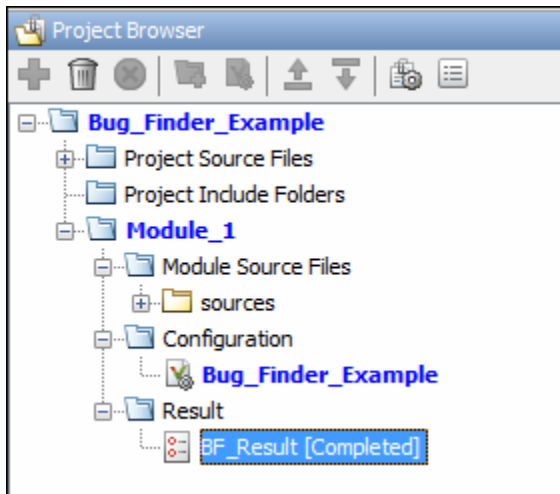
If compilation errors occur, the analysis continues on the remaining files that do compile. The **Dashboard** pane shows that some files did not compile and links to the **Output Summary** pane for details. The **Output Summary** pane shows compilation errors with a  icon.

For further diagnosis, select the error message for more details. Identify the line in your code responsible for the compilation error. You can use the error message details to understand why the line compiled with your compiler and what additional information Polyspace requires to emulate your compiler. See if you can work around the error by using a Polyspace option. For more information, see "Troubleshoot Compilation Errors".

For more precise run-time error checking in Code Prover, it is recommended that you fix all compilation errors. Use the option `Stop analysis if a file does not compile (-stop-if-compile-error)`.

Open Results

After analysis, the results open automatically. To open results that you have closed, double-click the result node on the **Project Browser** pane.



The Bug Finder (Code Prover) results are stored in a .psbf (.pscp) file in the results folder. For instance, if you save your project in C:\Projects\, a .psbf file for the Bug Finder analysis results on the first module Module_1 is stored in C:\Projects\Module_1\BF_Result. See also “Contents of Polyspace Project and Results Folders” on page 2-7.

See Also

More About

- “Run Polyspace Analysis from Command Line” on page 4-2
- “Run Polyspace Analysis by Using MATLAB Scripts” on page 5-9
- “Review Polyspace Bug Finder Results in Polyspace User Interface”
- “Upload Results to Polyspace Access” on page 2-25

Storage of Temporary Files During Polyspace Analysis

Polyspace produces some temporary files when performing an analysis. If your analysis runs slow or you encounter errors such as running out of disk space, check your temporary file location. For more information on possible errors, see:

- “Fix Polyspace Errors Related to Temporary Files” on page 32-61
- “Reduce Memory Usage and Time Taken by Polyspace Analysis” (Polyspace Code Prover)

To determine where to store temporary files, Polyspace looks for these environment variables in the following order:

- `RTE_TMP_DIR`: Define this environment variable only if you want to store Polyspace temporary files in a folder different from the standard temporary folders (defined by `TMPDIR` and such). You can see the current standard temporary folder by using the MATLAB® function `tempdir`.

Note This path must be an absolute path to an existing folder on which the current user has access rights (for reading and writing).

- `TMPDIR`
- `TMP`
- `TEMP`

If one of these variables is defined, Polyspace uses that path for storing temporary files. If these environment variables are not defined, Polyspace stores temporary files in:

- `/tmp` on Linux and Mac
- Folder specified with the `USERPROFILE` environment variable, folder returned from `GetWindowsDirectoryW` Windows API, or `Temp` directory on Windows

Run Polyspace Analysis with Windows or Linux Scripts

- “Run Polyspace Analysis from Command Line” on page 4-2
- “Modularize Polyspace Analysis by Using Build Command” on page 4-5
- “Select Files for Polyspace Analysis Using Pattern Matching” on page 4-11
- “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 4-15

Run Polyspace Analysis from Command Line

To run an analysis from a DOS or UNIX® command window, use the command `polyspace-bug-finder` or `polyspace-code-prover` followed by other options you wish to use. See also:

- `polyspace-bug-finder`
- `polyspace-code-prover`

To save typing the full path to the commands, add the path `polyspaceroot\polyspace\bin` to the `Path` environment variable on your operating system. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2023a`. See also “Installation Folder”.

Specify Sources and Analysis Options Directly

At the Windows, Linux or Mac OS X command-line, append sources and analysis options to the `polyspace-bug-finder` or `polyspace-code-prover` command.

For instance:

- To specify sources, use the `-sources` option followed by a comma-separated list of sources.

```
polyspace-bug-finder -sources C:\mySource\myFile1.c,C:\mySource\myFile2.c
```

If your current folder contains a `sources` subfolder with the source files, you can omit the `-sources` flag. The analysis considers files in `sources` and all subfolders under `sources`.

- To specify the target processor, use the `-target` option. For instance, to specify the `m68k` processor for your source file `file.c`, use the command:

```
polyspace-bug-finder -sources "file.c" -lang c -target m68k
```

- To check for violation of MISRA C™ rules, use the `-misra2` option. For instance, to check for only the required MISRA C rules on your source file `file.c`, use the command:

```
polyspace-bug-finder -sources "file.c" -misra2 required-rules
```

- To specify a results folder, use the option `-results-dir`.

Note that by default, the results folder is cleaned up and repopulated at each run. To avoid accidental removal of files during the cleanup, instead of using an existing folder that contains other files, specify a dedicated folder for the Polyspace results.

For the full list of analysis options, see:

- “Complete List of Polyspace Bug Finder Analysis Engine Options”
- “Complete List of Polyspace Code Prover Analysis Options” (Polyspace Code Prover)

For the full list of options, enter the following at the command line:

```
polyspace-bug-finder -help
```

Specify Sources and Analysis Options in Text File

Instead of specifying the options directly, you can save the options in a text file and use the text file each time you run the analysis.

- 1 Create an options file called `listoptions.txt` with your options. For example:

```
#These are the options for MyCodeProverProject
-lang c
-prog MyCodeProverProject
-author jsmith
-sources "mymain.c,funAlgebra.c,funGeometry.c"
-target x86_64
-compiler generic
-dos
-misra2 required-rules
-do-not-generate-results-for all-headers
-main-generator
-results-dir C:\Polyspace\MyCodeProverProject
```

- 2 Run Polyspace using options in the file `listoptions.txt`.

```
polyspace-code-prover -options-file listoptions.txt
```

See also `-options-file`.

Create Options File from Build System

If you use a build command (makefile) to build your source code, you can collect the sources and compiler options from your build command. Trace your build command to generate a text file with the required Polyspace options.

- 1 Create a list of Polyspace options using the configuration tool.

```
polyspace-configure -output-options-file \
    myOptions buildCommand
```

where *buildCommand* is the command you use to build your source code, for instance `make -B`.

See also `polyspace-configure`.

- 2 Run Polyspace using the options read from your build.

```
polyspace-bug-finder -options-file myOptions \
    -results-dir myResults
```

In addition to the options collected from your build command, you might want to add further options, for instance, to specify the defect checkers. You can append these options to the options file, add them directly at the command line or add them through a second options file (using another `-options-file` flag).

- 3 Open the results in the Polyspace user interface.

```
polyspace-bug-finder myResults
```

See Also

`polyspace-configure` | `polyspace-bug-finder` | `polyspace-code-prover`

More About

- “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 4-15

- “Modularize Polyspace Analysis by Using Build Command” on page 4-5

External Websites

- Set up Continuous Code Verification with Jenkins

Modularize Polyspace Analysis by Using Build Command

To configure the Polyspace analysis, you can reuse the compilation options in your build command such as `make`. First, you trace your build command with `polyspace-configure` (or `polyspaceConfigure` in MATLAB) and create a Polyspace options file. You later specify this options file for the subsequent Polyspace analysis.

If your build command creates several binaries, by default `polyspace-configure` groups the source files for all binaries into one Polyspace options file. If binaries that use the same source files or functions are compiled with different options, you lose this distinction in the subsequent Polyspace analysis. The presence of the same function multiple times can lead to link errors during the Polyspace analysis and sometimes to incorrect results.

This topic shows how to create a separate Polyspace options file for each binary created in your makefile. Suppose that a makefile creates four binaries: two executable (target `cmd1` and `cmd2`) and two shared libraries (target `liba` and `libb`). You can create a separate Polyspace options file for each of these binaries.

To try this example, use the files in `polyspaceroot\help\toolbox\bugfinder\examples\multiple_modules`. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2023a` or `C:\Program Files\Polyspace Server\R2023a`.

Build Source Code

Inspect the makefile. The makefile creates four binaries:

```
CC := gcc

LIBA_SOURCES := $(wildcard src/liba/*.c)
LIBB_SOURCES := $(wildcard src/libb/*.c)
CMD1_SOURCES := $(wildcard src/cmd1/*.c)
CMD2_SOURCES := $(wildcard src/cmd2/*.c)
LIBA_OBJ := $(notdir $(LIBA_SOURCES:.c=.o))
LIBB_OBJ := $(notdir $(LIBB_SOURCES:.c=.o))
CMD1_OBJ := $(notdir $(CMD1_SOURCES:.c=.o))
CMD2_OBJ := $(notdir $(CMD2_SOURCES:.c=.o))
LIBB_SOBJ := libb.so
LIBA_SOBJ := liba.so

all: cmd1 cmd2

cmd1: liba libb
    $(CC) -o $@ $(CMD1_SOURCES) $(LIBA_SOBJ) $(LIBB_SOBJ)

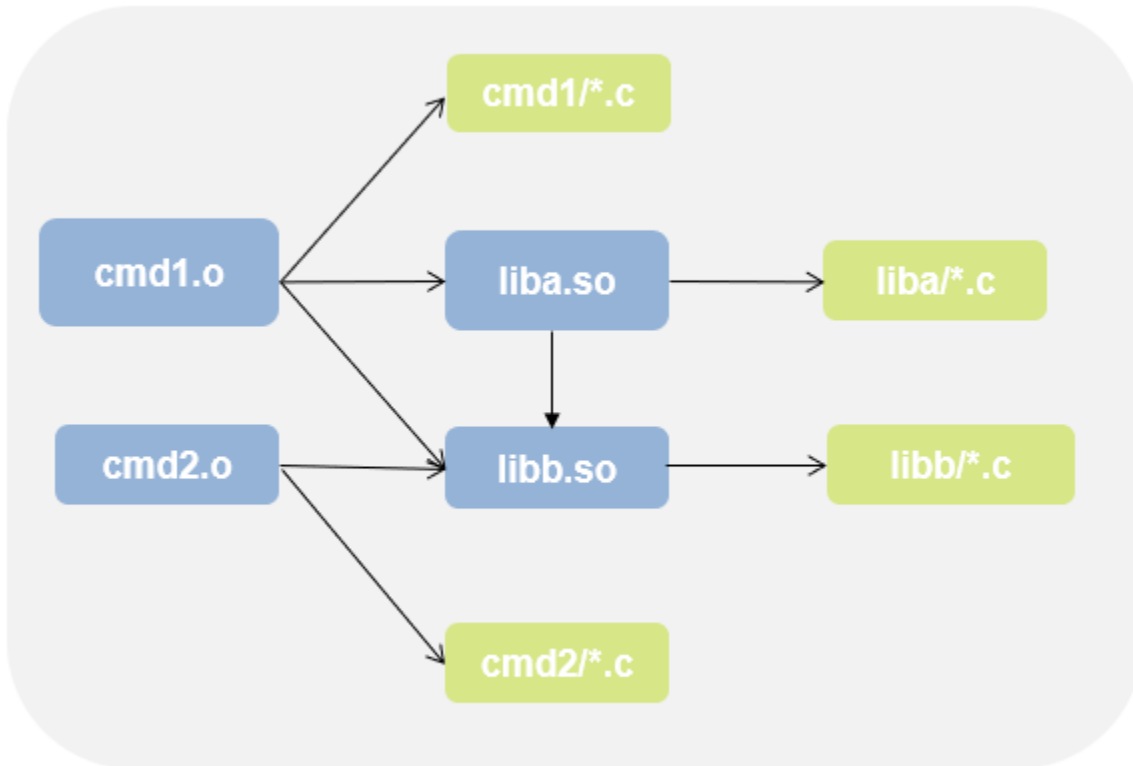
cmd2: libb
    $(CC) -c $(CMD2_SOURCES)
    $(CC) -o $@ $(CMD2_OBJ) $(LIBB_SOBJ)

liba: libb
    $(CC) -fPIC -c $(LIBA_SOURCES)
    $(CC) -shared -o $(LIBA_SOBJ) $(LIBA_OBJ) $(LIBB_SOBJ)

libb:
    $(CC) -fPIC -c $(LIBB_SOURCES)
    $(CC) -shared -o $(LIBB_SOBJ) $(LIBB_OBJ)

.PHONY: clean
clean:
    rm *.o *.so
```

The binaries created have the dependencies shown in this figure. For instance, creation of the object `cmd1.o` depends on all `.c` files in the folder `cmd1` and the shared objects `liba.so` and `libb.so`.



Build your source code by using the makefile. Use the `-B` flag to ensure full build.

```
make -B
```

Make sure that the build runs to completion.

Create One Polyspace Options File for Full Build

Trace the build command by using `polyspace-configure`. Use the option `-output-options-file` to create a Polyspace options file `psoptions` from the build command.

```
polyspace-configure -output-options-file psoptions make -B
```

Run Bug Finder or Code Prover by using the previously created options file: Save the analysis results in a `results` subfolder.

```
polyspace-bug-finder -options-file psoptions -results-dir results
```

You see this link error (warning in Bug Finder):

```
Procedure 'main' multiply defined.
```

The error occurs because the files `cmd1/cmd1_main.c` and `cmd2/cmd2_main.c` both have a `main` function. When you run your build command, the two files are used in separate targets in the makefile. However, `polyspace-configure` by default creates one options file for the full build. The Polyspace options file contains both source files resulting in conflicting definitions of the `main` function.

To verify the cause of the error, open the Polyspace options file `psoptions`. You see these lines that include the files with conflicting definitions of the `main` function.

```
-sources src/cmd1/cmd1_main.c
-sources src/cmd2/cmd2_main.c
```

Create Options File for Specific Binary in Build Command

To avoid the link error, build the source code for a specific binary when tracing your build command by using `polyspace-configure`.

For instance, build your source code for the binary `cmd1.o`. Specify the makefile target `cmd1` for `make`, which creates this binary.

```
polyspace-configure -output-options-file psoptions -allow-overwrite make -B cmd1
```

Run Bug Finder or Code Prover by using the previously created options file.

```
polyspace-bug-finder -options-file psoptions -results-dir results
```

The link error does not occur and the analysis runs to completion. You can open the Polyspace options file `psoptions` and see that only the source files in the `cmd1` subfolder and the files involved in creating the shared objects are included with the `-sources` option. The source files in the `cmd2` subfolder, which are not involved in creating the binary `cmd1.o`, are not included in the Polyspace options file.

Special Considerations for Libraries (Code Prover only)

If you trace the creation of a shared object from libraries, the source files extracted do not contain a `main` function. In the subsequent Code Prover analysis, you can see an error because of the missing `main`.

Use the Polyspace option `Verify module or library (-main-generator)` to generate a `main` function. Specify the option in the options file that was created or directly at the command line. See “Verify C Application Without main Function” (Polyspace Code Prover).

In C++, use these additional options for classes:

- `Class (-class-analyzer)`
- `Functions to call within the specified classes (-class-analyzer-calls)`

Create One Options File Per Binary Created in Build Command

To create an options file for a specific binary created in the build command, you must know the details of your build command. If you are not familiar with the internal details of the build command, you can create a separate Polyspace options file for *every* binary created in the build command. The approach works for binaries that are executables, shared (dynamic) libraries and static libraries.

This approach works only if you use these compilers:

- GNU C or GNU C++
- Microsoft Visual C++

Trace the build command by using `polyspace-configure`. To create a separate options file for each binary, use the option `-module` with `polyspace-configure`.

```
polyspace-configure -module -output-options-path optionsFilesFolder make -B
```

The command creates options files in the folder `optionsFilesFolder`. In the preceding example, the command creates four options files for the four binaries:

- `cmd1.psopts`
- `cmd2.psopts`
- `liba_so.psopts`
- `libb_so.psopts`

You can run Polyspace on the code implementation of a specific binary by using the corresponding options file. For instance, you can run Code Prover on the code implementation of the binary created from the makefile target `cmd1` by using this command:

```
polyspace-bug-finder -options-file optionsFilesFolder\cmd1.psopts -results-dir results
```

For this approach, you do not need to know the details of your build command. However, when you create a separate options file for each binary in this way, each options file contains source files directly involved in the binary and not through shared objects. For instance, the options file `cmd1.psopts` in this example specifies only the source files in the `cmd1` subfolder and not the source files involved in creating the shared objects `liba.so` and `libb.so`. The subsequent analysis by using this options file cannot access functions from the shared objects and uses function stubs instead. In the Code Prover analysis, if you see too many orange checks due to the stubbing, use the approach stated in the section “Create Options File for Specific Binary in Build Command” on page 4-8.

Special Considerations for Libraries (Code Prover only)

If you trace the creation of a shared object from libraries, the source files extracted do not contain a main function. In the subsequent Code Prover analysis, you can see an error because of the missing main.

Use the Polyspace option `Verify module` or `library` (`-main-generator`) to generate a main function. Specify the option in the options file that was created or directly at the command line. See “Verify C Application Without main Function” (Polyspace Code Prover).

In C++, use these additional options for classes:

- `Class` (`-class-analyzer`)
- Functions to call within the specified classes (`-class-analyzer-calls`)

See Also

`polyspace-configure` | `polyspace-bug-finder` | `polyspace-bug-finder-server`

More About

- “Run Polyspace Analysis from Command Line” on page 4-2
- “Create Polyspace Analysis Configuration from Build Command (Makefile)” on page 13-22

Select Files for Polyspace Analysis Using Pattern Matching

When you run static analysis using Polyspace products, the analysis covers all files specified in your Polyspace project (or specified using `-sources` at the command line). Sometimes, you might want to see results only in a subset of these files, or might want a different analysis behavior to apply to a subset of files. You can specify a subset of files using file selection patterns. The file selection patterns (glob patterns) use wildcards such as `?` or `*` to cover multiple files.

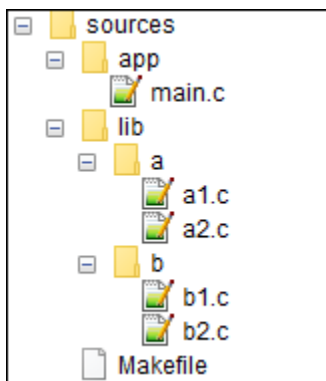
When to Specify File Selection Patterns

You can select a subset of files when creating a Polyspace project or options file from your build command, or when running static analysis using Polyspace Bug Finder.

Select Files When Setting Up Polyspace Analysis from Build Command

When you create projects by using `polyspace-configure`, you can include or exclude source files whose paths match the pattern that you pass to the options `-include-sources` or `-exclude-sources`. You can specify these two options multiple times and combine them at the command line.

This folder structure applies to these examples.



To try these examples, use the demo files in `polyspaceroot\help\toolbox\bugfinder\examples\sources-select`. `polyspaceroot` is the Polyspace installation folder.

Run this command:

```
polyspace-configure -allow-overwrite -include-sources "glob_pattern" \
-print-excluded-sources -print-included-sources make -B
```

`glob_pattern` is the glob pattern that you use to match the paths of the files you want to include or exclude from your project. To ensure the shell does not expand the glob patterns you pass to `polyspace-configure`, enclose them in double quotes.

Select Files When Running Bug Finder Analysis

When analyzing C/C++ code with Polyspace Bug Finder, you can define file sets in your project that need specific treatment during analysis. For instance, you might want to skip the definitions of function bodies in third-party libraries or force analysis of all functions in files that you own. You can enumerate file sets with specific behaviors in a classification XML file and fine-tune the Bug Finder analysis using this classification file.

In the classification XML file, you can specify file patterns inside a `file-pattern` element (child of `fileset > files-in-set` or `fileset > files-not-in-set` element). For instance, the following patterns select `.hpp` files in subfolders of `myproject/inc` but excludes files ending with `-generated`.

```
<fileset name="Application implementation and header files">
  <files-in-set>
    <file-pattern>myproject/inc/**/*\.hpp</file-pattern>
  </files-in-set>
  <files-not-in-set>
    <file-pattern>myproject/inc/**/*-generated.hpp</file-pattern>
  </files-not-in-set>
  <behaviors>
    <!-- Specific behaviors for this file set -->
  </behaviors>
</fileset>
```

To specify a classification file during static analysis, use the analysis option `-classification`. For instance, you can run Bug Finder using this command:

```
polyspace-bug-finder -options-file myOptions.txt -classification myClassification.xml
```

For more information, see:

- `-classification`
- “Classify Project Files Into File Sets for Precise Control of Bug Finder Analysis” on page 19-2

Supported Patterns for File Selection

In the table, the examples assume that `sources` is a top-level folder.

Glob Pattern Syntax	Example
No special characters, slashes ('/'), or backslashes ('\'). Pattern matches corresponding files, but not folders.	<code>-include-sources "main.c"</code> matches: <code>/sources/app/main.c</code>
Pattern contains '*' or '?' special characters. '*' matches zero or more characters in file or folder name. '?' matches one character in file or folder name. The matches do not include path separators.	<code>-include-sources "b?.c"</code> matches: <code>/sources/lib/b/b1.c</code> <code>/sources/lib/b/b2.c</code> <code>-include-sources "app/*.c"</code> matches: <code>/sources/app/main.c</code>
Pattern starts with: • A slash '/' (UNIX). • Drive letter, for example C:\ (Windows). Pattern matches absolute path only.	<code>-include-sources "/a"</code> does not match anything. <code>-include-sources "/sources/app"</code> matches: <code>/sources/app/main.c</code>

Glob Pattern Syntax	Example
Pattern ends with: <ul style="list-style-type: none"> • A slash (UNIX). • A backslash (Windows). • A double asterisk ('**') Pattern matches all files under specified folder. '**' is ignored if it is at the start of the pattern.	<pre>-include-sources "a/" matches /sources/lib/a/a1.c /sources/lib/a/a2.c</pre>
Pattern contains: <ul style="list-style-type: none"> • '/**/' (UNIX). • '**\' (Windows). Pattern matches zero or more folders in the specified path.	<pre>-include-sources "lib/**/?1.c" matches: /sources/lib/a/a1.c /sources/lib/b/b1.c</pre>
Pattern starts with '.' or '..'. Pattern matches paths relative to the path where you run the command.	<pre>If you start polyspace-configure from / sources/lib/a, -include-sources "../lib/**/b?.c" matches: /sources/lib/b/b1.c /sources/lib/b/b2.c</pre>
Pattern is a UNC path on Windows .	<pre>If your files are on server myServer: \\myServer\sources\lib\b** matches: \\myServer\sources\lib\b\b1.c \\myServer\sources\lib\b\b2.c</pre>

polyspace-configure does not support these glob patterns:

- Absolute paths relative to the current drive on Windows.
For instance, \foo\bar.
- Relative paths to the current folder.
For instance, C:foo\bar.
- Extended length paths in Windows.
For instance, \\?\foo.
- Paths that contain '.' or '..' except at the start of the pattern.
For instance, /foo/bar/./a?.c.
- The '*' character by itself.

See Also

-classification | polyspace-configure

More About

- “Classify Project Files Into File Sets for Precise Control of Bug Finder Analysis” on page 19-2

Configure Polyspace Analysis Options in User Interface and Generate Scripts

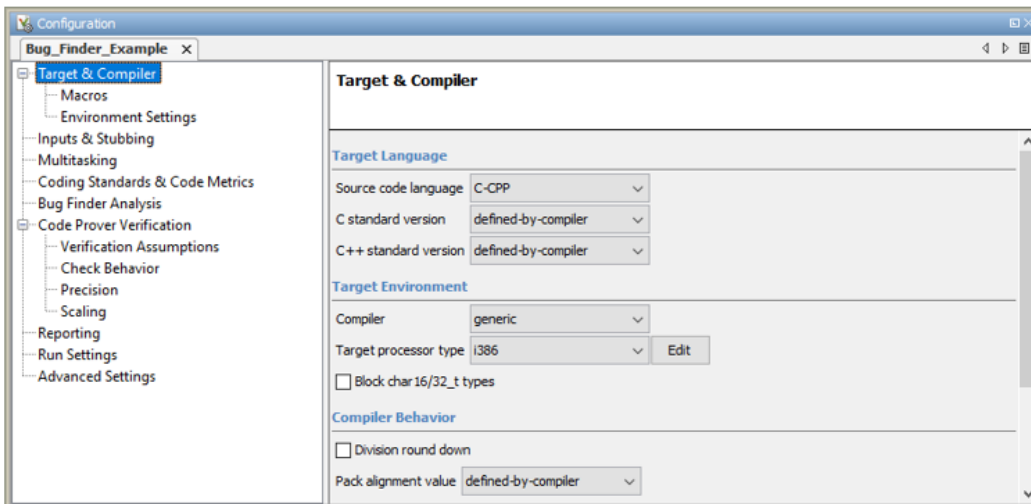
In this section...

“Prerequisites” on page 4-16

“Generate Scripts from Configuration” on page 4-16

“Run Analysis with Generated Scripts” on page 4-17

If you have an installation of the desktop products, Polyspace Bug Finder and/or Polyspace Code Prover, you can configure your project in the user interface of the desktop products. You can then generate a script or an options file from the configuration defined in the user interface and use the script or options file for automated runs with the desktop or server products.



```
polyspace -generate-launching-script-for Bug_Finder_Example.psrpj -bug-finder
polyspace -generate-launching-script-for Code_Prover_Example.psrpj
```

```
-target x86_64
-c-version c11
-compiler gnu4.6
-dos
-sources-list-file source_command.txt
...
```

Unless you create a Polyspace project from existing specifications such as a build command, when setting up the project, you might have to perform a few trial runs first. In these trial runs, if you run into compilation errors or unchecked code, you might have to modify your analysis configuration. It is easier performing this initial setup in the user interface of the desktop products. The user interface provides various features such as:

- Auto-generation of XML file for constraint specification.
- Context-sensitive help for options.

Prerequisites

You must have at least one license of Polyspace Bug Finder and/or Polyspace Code Prover to open the Polyspace user interface and configure the options.

After generating the scripts, you can run the analysis using either the desktop products (Polyspace Bug Finder and Polyspace Code Prover) or the server products (Polyspace Bug Finder Server and/or Polyspace Code Prover Server).

Generate Scripts from Configuration

This example shows how to generate a script from a Bug Finder configuration. The same steps apply to a Code Prover configuration.

- 1 Add source files to a new project in the Polyspace user interface.

Navigate to *polyspaceroot*\polyspace\bin, where *polyspaceroot* is the Polyspace installation folder, for instance, C:\Program Files\Polyspace\R2023a. Open the Polyspace user interface using the *polyspace* executable and create a new project.

See “Add Source Files for Analysis in Polyspace Desktop User Interface” on page 2-2.

- 2 Specify the analysis options on the **Configuration** pane in the Polyspace project. To open this pane, in the project browser, click the configuration node in your Polyspace project.

See “Specify Polyspace Analysis Options” on page 12-2.

- 3 Run the analysis. Based on compilation errors and analysis results, modify options as needed.

See “Run Analysis in Polyspace Desktop User Interface” on page 3-2.

- 4 Once your analysis options are set, generate a script from the project (.psprj file).

To generate a script from the demo project, *Bug_Finder_Example*:

- a Load the project. Select **Help > Examples > Bug_Finder_Example.psprj**. A copy of this project is loaded in the *Examples* folder in your default workspace. To find the project location, place your cursor on the project name in the **Project Browser** pane.
- b Navigate to the project location and enter:

```
polyspace -generate-launching-script-for Bug_Finder_Example.psprj -bug-finder
```

To generate Code Prover scripts, use the same command without the `-bug-finder` option.

If a project has more than one module (with more than one configuration in each module), the options from the currently active configuration in the currently active module will be extracted in the script.

These files are generated for scripting the analysis:

- `source_command.txt`: Lists source files. This file can be provided as argument to the `-sources-list-file` option.

- `options_command.txt`: Lists analysis options. This file can be provided as argument to the `-options-file` option.
- `launchingCommand.bat` or `launchingCommand.sh`, depending on your operating system. The file uses the `polyspace-bug-finder` or `polyspace-code-prover` executable to run the analysis. The analysis runs on the source files listed in `source_command.txt` and uses the options listed in `options_command.txt`.

Run Analysis with Generated Scripts

After configuring your analysis and generating scripts, you can use the generated files to automate the subsequent analysis. You can automate the subsequent analysis using either the desktop or server products.

To automate a Bug Finder analysis with the desktop product, Polyspace Bug Finder:

- 1 Generate scripts as mentioned in the previous section.
- 2 Execute the script `launchingCommand.bat` or `launchingCommand.sh` at periodic intervals or based on predefined triggers.

To automate a Bug Finder analysis with the server product, Polyspace Bug Finder Server:

- 1 After specifying options in the user interface and before generating scripts, move the Polyspace project (`.psprj` file) to the server where the server product is running.
- 2 Generate scripts as mentioned in the previous section.

The scripts refer to the server product executable instead of the desktop products.

- 3 Execute the script `launchingCommand.bat` or `launchingCommand.sh` at periodic intervals or based on predefined triggers.

Alternatively, you can modify the script generated for the desktop product so that the server product is executed. The script refers to the path to a desktop product executable, for instance:

```
"C:\Program Files\Polyspace\R2023a\polyspace\bin\polyspace-code-prover.exe"
```

Replace this with the path to a server product executable, for instance:

```
"C:\Program Files\Polyspace Server\R2023a\polyspace\bin\
  polyspace-code-prover-server.exe"
```

Sometimes, you might want to override some of the options in the options file. For instance, the option to specify a results folder is hardcoded in the script. You can remove this option or override it when launching the scripts:

```
launchingCommand -results-dir newResultsFolder
```

where *newResultsFolder* is the new results folder. This folder can even be dynamically generated for each run.

If you override multiple options in `options_command.txt`, you can save the overrides in a second options file. Modify the script `launchingCommand.bat` or `launchingCommand.sh` so that both options files are used. The script uses the option `-options-file` to use an options file, for instance:

```
-options-file options_command.txt
```

If you place your option overrides in a second options file `overrides.txt`, modify the script to append a second `-options-file` option:

```
-options-file options_command.txt -options-file overrides.txt
```

See Also

`-generate-launching-script-for`

Related Examples

- “Run Polyspace Analysis from Command Line” on page 4-2
- “Run Polyspace Bug Finder on Server and Upload Results to Web Interface”

Run Polyspace Analysis with MATLAB Scripts

- “Integrate Polyspace with MATLAB and Simulink” on page 5-2
- “Get Started with Polyspace Analysis by Using MATLAB” on page 5-5
- “Run Polyspace Analysis by Using MATLAB Scripts” on page 5-9
- “Compare Results from Different Polyspace Runs by Using MATLAB Scripts” on page 5-13
- “Generate MATLAB Scripts from Polyspace User Interface” on page 5-16
- “Troubleshoot Polyspace Analysis from MATLAB” on page 5-18

Integrate Polyspace with MATLAB and Simulink

Polyspace Bug Finder and Polyspace Code Prover are standalone products. Install these Polyspace products by using the MathWorks® installer. See “Install Polyspace with Other MathWorks Products”.

Polyspace products are installed in a different root folder from other MathWorks products. For instance, in Windows:

- The default MATLAB root folder is C:\Program Files\MATLAB\R2023a.
- The default Polyspace root folder is C:\Program Files\Polyspace\R2023a.

To run Polyspace from MATLAB, Simulink, or MATLAB Coder™, perform a post-installation procedure to integrate Polyspace with MATLAB and Simulink.

The integration process and supported MATLAB releases might be different for previous Polyspace releases. Check the documentation of your release if you have Polyspace from an older release.

Same Release of Polyspace and MATLAB

If Polyspace and MATLAB are both from the same release, you can do the following after integrating Polyspace and MATLAB:

- Run a Polyspace analysis on C/C++ code generated from a model or included as custom code in a model from the Simulink Editor. You can also run these analyses using a MATLAB script. See “Bug Finder Analysis in Simulink”.
- If you have Embedded Coder®, run a Polyspace analysis on C/C++ code that is generated from MATLAB code by using the MATLAB Coder App. See “Bug Finder Analysis in MATLAB Coder”.
- Run a Polyspace analysis on hand-written C/C++ code by using MATLAB scripts. See “Bug Finder Analysis with MATLAB Scripts”.

Note that the MATLAB-Polyspace integration does not make the Polyspace documentation available within the MATLAB Help Browser. You can continue to access the Polyspace documentation online.

Prerequisite

Before you integrate Polyspace with MATLAB or Simulink from the same release, determine if your MATLAB or Simulink is already integrated with Polyspace. See “Check Integration Between MATLAB and Polyspace” on page 5-4.

Integrate Polyspace with MATLAB or Simulink

- 1 Open MATLAB with administrator or root privileges. For instance, in Windows, to open MATLAB with administrator privilege, right-click the MATLAB executable and select **Run as administrator**.
- 2 At the MATLAB command prompt, enter the following:

```
polyspacesetup('install');
```

If you installed Polyspace in the default folder C:\Program Files\Polyspace\R2023a, the command integrates Polyspace with MATLAB. If a Polyspace installation is not detected at the default location, you are prompted for the installation location. Alternatively, use:

```
polyspacesetup('install','polyspaceFolder',Folder)
```

where *Folder* is the Polyspace installation folder. If you are prompted that the workspace will be cleared and that all open models closed, click **Yes**. The process might take a few minutes to complete. To avoid interactive prompts, enter:

```
polyspacesetup('install', 'polyspaceFolder', Folder, 'silent', true);
```

- 3 Restart MATLAB.

You can also perform the integration by using a script. See “Integrate Polyspace Noninteractively with MATLAB at Command Line by Using `-batch`”.

Unlink and Relink MATLAB and Polyspace

You can integrate MATLAB with only one instance of Polyspace. To integrate with a different instance of Polyspace, uninstall the current integration. At the MATLAB command prompt, enter:

```
polyspacesetup('uninstall')
```

This step uninstalls only the integration between MATLAB and Polyspace. To uninstall an instance of Polyspace, use the MathWorks installer.

MATLAB Release Earlier Than Polyspace

You can also integrate Polyspace with MATLAB or Simulink from an earlier release. This cross-release integration offers limited functionalities compared to the same-release integration. In a cross-release workflow:

- You can run a Polyspace analysis of generated C/C++ code in the MATLAB Command Window.
- You cannot analyze custom code included in models or handwritten code.
- You cannot start Polyspace analyses from the Simulink Editor or MATLAB Coder App.

See “Polyspace Support of MATLAB and Simulink from Different Releases” on page 6-68.

Prerequisite

To perform a cross-release integration, these conditions must be true:

- The MATLAB or Simulink release supports cross-release integration with a Polyspace release. See “Polyspace Support of MATLAB and Simulink from Different Releases” on page 6-68.
- MATLAB or Simulink is not already integrated with Polyspace. To determine if Polyspace is already integrated, see “Check Integration Between MATLAB and Polyspace” on page 5-4.

Integrate Polyspace with Cross-Release MATLAB or Simulink

- 1 Open MATLAB.
- 2 At the MATLAB command prompt, enter:

```
polyspacesetup('install', 'polyspaceFolder', Folder)
```

where *FOLDER* is the Polyspace installation folder. If you are prompted that the workspace will be cleared and that all open models closed, click **Yes**. The process might take a few minutes to complete. To avoid interactive prompts, enter:

```
polyspacesetup('install', 'polyspaceFolder', Folder, 'silent', true);
```

- 3 Restart MATLAB. This integration process does not integrate the Polyspace documentation with the MATLAB Help Browser.

In addition to using a command line prompt, you can also perform the integration by using a script. See “Integrate Polyspace Noninteractively with MATLAB at Command Line by Using -batch”.

You can integrate MATLAB with only one instance of Polyspace. To integrate with a different instance of Polyspace, uninstall the current integration. At the MATLAB command prompt, enter:

```
polyspacesetup('uninstall')
```

This step uninstalls only the integration between MATLAB and Polyspace. To uninstall an instance of Polyspace, use the MathWorks installer.

Check Integration Between MATLAB and Polyspace

To determine if MATLAB is already linked to Polyspace, open MATLAB and enter:

```
ver
```

If Polyspace is integrated with MATLAB, you see the Polyspace products in the list of installed products.

The integration of MATLAB and Polyspace adds Polyspace installation subfolders to the MATLAB search path. To see the added paths, enter:

```
polyspacesetup('showpolyspacefolders')
```

See Also

`polyspacesetup`

More About

- “Polyspace Support of MATLAB and Simulink from Different Releases” on page 6-68
- “Bug Finder Analysis with MATLAB Scripts”
- “Bug Finder Analysis in Simulink”
- “Bug Finder Analysis in MATLAB Coder”
- “Fix Issues When when Integrating Polyspace with MATLAB and Simulink” on page 32-65

Get Started with Polyspace Analysis by Using MATLAB

This tutorial shows how to analyze handwritten C/C++ code by running a Polyspace analysis from the MATLAB Command Window or the MATLAB Editor. To analyze code generated from a Simulink model, see “Run Polyspace Analysis on Code Generated from Simulink Model” on page 6-15.

Prerequisites

Integrate Polyspace with MATLAB before you run a Polyspace analysis from the MATLAB Command Window. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

Run Polyspace Analysis by Using MATLAB

You analyze handwritten C code by configuring and then starting a Polyspace analysis from the MATLAB Command Window or the MATLAB Editor.

To perform a Polyspace analysis, create a `polyspace.Project` object, specify the source files and the analysis options, and then start the analysis by using this object. To create a `polyspace.Project` object, use the function `polyspace.Project`.

```
psPrj = polyspace.Project;
```

In this tutorial, the handwritten code in the file `numerical.c` is analyzed. The file `numerical.c` is part of your Polyspace software. This source file and the header files required to analyze it can be found in the folder `polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources`. Here, `polyspaceroot` is the location of the Polyspace installation folder in your development environment. Create the paths to these source and header files by using the function `fullfile`.

```
% Create the Path to source and header files
sourceFile = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c');
includeFolder = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');
```

Associate the source and header files with the `psPrj` object.

```
% Associate the source and header files
psPrj.Configuration.Sources = {sourceFile};
psPrj.Configuration.EnvironmentSettings.IncludeFolders = {includeFolder};
```

Configure the Polyspace analysis options. For instance, you can specify the compiler for the Polyspace analysis and check for violation of specific coding rules. You can also specify a folder where you store the generated results. For instance, store the results in the folder 'results' in the current working directory.

```
% Specify target compiler
psPrj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
% Enable Mchecking for MISRA C violation
psPrj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
psPrj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = 'mandatory';
% Specify results folder
psPrj.Configuration.ResultsDir = fullfile(pwd, 'results');
```

The variable `pwd` contains the path of the current working directory. For details on configurable Polyspace analysis options, see `polyspace.Project.Configuration` Properties.

Start the Polyspace analysis by using the function `run`.

```
% start BugFinder analysis
bfStatus = run(psPrj, 'bugFinder');
```

The progress of the Polyspace analysis appears in the MATLAB Command Window. When the analysis is successful, `bfStatus` is set to `0`.

The Polyspace analysis result consists of a list of Bug Finder defects. To view a summary of the Bug Finder defects in a MATLAB table, use the function `getSummary`. For more details about obtaining summary of different kinds of results, see `getSummary`.

```
% Obtain list of Bug Finder defects
resObj = psPrj.Results;
bfSummary = getSummary(resObj, 'defects');
```

The Bug Finder defects are listed in the 9x4 table `bfSummary`.

9x4 [table](#)

Category	Defect	Impact	Total
Numerical	Absorption of float operand	High	1
Numerical	Float conversion overflow	High	1
Numerical	Float division by zero	High	1
Numerical	Integer conversion overflow	High	1
Numerical	Integer division by zero	High	1
Numerical	Invalid use of standard library floating point routine	High	1
Numerical	Invalid use of standard library integer routine	High	2
Numerical	Sign change integer conversion overflow	Medium	1
Numerical	Unsigned integer conversion overflow	Low	1

Frequently Used MATLAB Functions

This table lists some MATLAB functions that you can use for automating a Polyspace analysis from the MATLAB Editor or Command Window.

Function	Application
<code>fopen</code>	Opens a file for binary read access. For instance, use this function to read an error log file.
<code>fclose</code>	Closes a file that was opened by using <code>fopen</code> . For instance, use this function to close an error log file after reading it.
<code>open</code>	Opens a file outside MATLAB in an appropriate application. For instance, use this function to open <code>psprj</code> files in the Polyspace UI.

Function	Application
<code>exist</code>	Checks for the existence of an entity. For instance, use this function to check if a particular folder or file already exists.
<code>delete</code>	Deletes a file or an object. For instance, use this function to delete older results or unnecessary options objects.
<code>questdlg</code>	Creates a configurable dialog box. Use this function to change different settings of a Polyspace analysis in a script. For instance, you can choose to enable different coding rules based on the output of this function.
<code>clear</code>	Clears the workspace by deleting all objects. You can this function at the beginning of the Polyspace analysis.
<code>clc</code>	Clears all text from the MATLAB Command Window.
<code>fullfile</code>	Builds full file names from its parts. For instance, use this function to construct the full paths to source files.
<code>char</code>	Converts an array to a character array. For instance, use this function to construct the input arguments to functions that take character arrays.
<code>string</code>	Converts a variable into string arrays. For instance, use this function to construct input arguments for functions that take strings.
<code>dir</code>	Lists the content of the current working folder. For instance, use this function to find specific files or folders in the current folder.
<code>system</code>	Executes operating system commands and returns their outputs. For instance, use this function to execute a command-line script without exiting MATLAB.
<code>disp</code>	Displays the value of the input variable. For instance, use this function for debugging code, similar to how <code>printf()</code> is used in C code.
<code>visdiff</code>	Compares two files or folder. For instance, use this function to compare results from different Polyspace analysis to see the difference.
<code>ismember</code>	Determines if the elements in one array are also present in another array. For instance, use this function to check if a checker or coding rule is enabled in a Polyspace analysis, or to filter results to find a specific check.
<code>any</code>	Determines if any array elements are nonzero. For instance, use this function to check for new results.
<code>nnz</code>	Returns the number of nonzero matrix elements. For instance, use this function to check for new results.
<code>fieldnames</code>	Reads a structure, a Java object, or a Microsoft COM object and returns the field names. For instance, use this function to read and manipulate tables.

See Also

`polyspace.Project` | `polyspaceBugFinder` | `run`

Related Examples

- “Run Polyspace Analysis by Using MATLAB Scripts” on page 5-9
- “Visualize Bug Finder Analysis Results in MATLAB” on page 24-11
- “Troubleshoot Polyspace Analysis from MATLAB” on page 5-18
- “Generate MATLAB Scripts from Polyspace User Interface” on page 5-16

Run Polyspace Analysis by Using MATLAB Scripts

You can automate the analysis of your C/C++ code by using MATLAB scripts. In your script, you specify your source files and analysis options such as compiler, run an analysis, and read the analysis results to MATLAB tables.

For instance, use this script to run a Polyspace Bug Finder analysis on a sample file:

```
proj = polyspace.Project

% Specify sources and includes
sourceFile = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c');
includeFolder = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');

% Configure analysis
proj.Configuration.Sources = {sourceFile};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.EnvironmentSettings.IncludeFolders = {includeFolder};
proj.Configuration.ResultsDir = fullfile(pwd, 'results');

% Run analysis
bfStatus = run(proj, 'bugFinder');

% Read results
resObj = proj.Results;
bfSummary = getSummary(resObj, 'defects');
```

See also `polyspace.Project`.

Prerequisites

Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

Specify Multiple Source Files

You can specify a folder containing all your source files. For instance, if `proj` is a `polyspace.Project` object, enter:

```
sourceFolder = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');
proj.Configuration.Sources = {fullfile(sourceFolder, '*')}
```

You can also specify multiple source folders in the cell array.

You can specify a folder that contains all your source files both directly *and in subfolders*. For instance:

```
sourceFolder = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');
proj.Configuration.Sources = {fullfile(sourceFolder, '**')}
```

If you do not want to analyze all files in a folder, you can explicitly specify which files to analyze. For instance:

```
sourceFolder = fullfile(polyspaceroot, 'polyspace', ...  
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');  
file1 = fullfile(sourceFolder, 'numerical.c');  
file2 = fullfile(sourceFolder, 'staticmemory.c');  
proj.Configuration.Sources = {file1, file2};
```

You can explicitly exclude files from analysis. For instance:

```
% Specify source folder.  
sourceFolder = fullfile(polyspaceroot, 'polyspace', ...  
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');  
proj.Configuration.Sources = {fullfile(sourceFolder, '**')};  
  
% Specify files to exclude.  
file1 = fullfile(sourceFolder, 'security.c');  
file2 = fullfile(sourceFolder, 'tainteddata.c');  
proj.Configuration.InputsStubbing.DoNotGenerateResultsFor = ['custom=' file1 ...  
    ', ' file2];
```

However, this method of exclusion does not apply to Code Prover run-time error checking.

Check for MISRA C:2012 Violations

You can customize the Polyspace analysis to check for MISRA C:2012 rule violations.

Set options for checking MISRA C:2012 rules. Disable the regular Bug Finder analysis, which looks for defects.

If `proj` is a `polyspace.Project` object, to run a Bug Finder analysis with all mandatory MISRA C:2012 rules, enter:

```
% Enable MISRA C checking  
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;  
proj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = 'mandatory';  
  
% Disable defect checking  
proj.Configuration.BugFinderAnalysis.EnableCheckers = false;  
  
% Run analysis  
bfStatus = run(proj, 'bugFinder');  
  
% Read summary of results  
resObj = proj.Results;  
misraSummary = getSummary(resObj, 'misraC2012');
```

Check for Specific Defects or Coding Rule Violations

Instead of the default set of defect or coding rule checkers, you can specify your own set.

If `proj` is a `polyspace.Project` object, to disable MISRA C:2012 rules 8.1 to 8.4, enter:

```
% Disable rules  
misraRules = polyspace.CodingRulesOptions('misraC2012');  
  
misraRules.Section_8_Declarations_and_definitions.rule_8_1 = false;  
misraRules.Section_8_Declarations_and_definitions.rule_8_2 = false;
```

```
misraRules.Section_8_Declarations_and_definitions.rule_8_3 = false;
misraRules.Section_8_Declarations_and_definitions.rule_8_4 = false;
```

```
% Configure analysis
```

```
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = misraRules;
```

See also `polyspace.CodingRulesOptions`.

To enable Bug Finder defects, use the class `polyspace.DefectsOptions`. One difference between coding rules and defects class is that coding rule checkers are enabled by default. You disable the ones that you do not want. All defect checkers are disabled by default. You enable the ones that you want.

You can also specify a coding standard XML file that enables coding rules from different standards. When checking for coding rule violations, you can refer to the file. For instance, to use the template XML file `StandardsConfiguration.xml` provided with the product in the subfolder `polyspace\examples\cxx\Bug_Finder_Example\sources`, enter:

```
pathToTemplate = fullfile(polyspaceroot, 'polyspace', 'examples', ...
    'cxx', 'Bug_Finder_Example', 'sources', 'StandardsConfiguration.xml');
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = 'from-file';
proj.Configuration.CodingRulesCodeMetrics.EnableCheckersSelectionByFile = true;
proj.Configuration.CodingRulesCodeMetrics.CheckersSelectionByFile = pathToTemplate;
```

Find Files That Do Not Compile

If one or more of your files contain a compilation error, the analysis continues with the remaining files. You can choose to stop analysis on compilation errors.

If `proj` is a `polyspace.Project` object, to stop analysis on compilation errors, enter:

```
proj.Configuration.EnvironmentSettings.StopWithCompileError = true;
```

However, it is more convenient to let the analysis complete and capture all compilation errors from the analysis log file. For more information, see “Troubleshoot Polyspace Analysis from MATLAB” on page 5-18.

Run Analysis on Server

You can run an analysis on a remote server instead of your local desktop. Once you have set up connection to a server, you can run the analysis in batch mode. For setup information, see “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

Specify that the analysis must run on a server. Specify a folder on your desktop where results are downloaded after analysis. If `proj` is a `polyspace.Project` object, to configure analysis on a server, enter:

```
proj.Configuration.MergedComputingSettings.BatchBugFinder = true;
proj.Configuration.ResultsDir = fullfile(pwd, 'results');
```

Specify the head node that manages the Polyspace jobs:

```
proj.Configuration.Advanced.Additional = '-scheduler nodeHost'
```

Run analysis as usual.

```
run(proj, 'bugFinder');
```

Open the results from the results folder location.

```
pslinkfun('openresults', '-resultsfolder', proj.Configuration.ResultsDir);
```

If the analysis is complete and the results have been downloaded, they open in the Polyspace user interface.

See Also

`polyspace.Project` | `polyspaceBugFinder` | `-scheduler`

Related Examples

- “Generate MATLAB Scripts from Polyspace User Interface” on page 5-16
- “Visualize Bug Finder Analysis Results in MATLAB” on page 24-11
- “Troubleshoot Polyspace Analysis from MATLAB” on page 5-18

Compare Results from Different Polyspace Runs by Using MATLAB Scripts

This topic shows how to run Polyspace by using MATLAB scripts, save each result in a separate folder, and see only new or unreviewed results compared to the last run.

If your project consists of legacy code, it is often beneficial to run a preliminary analysis. In the subsequent runs, you can focus only on results related to newly added code.

Review Only New Results Compared to Last Run

To see only new results, specify that the current run must import results and comments from the results folder of the last run.

This script saves results of each Polyspace run in a separate folder and compares each result set with the result set from the previous run.

- The first time you run the script, all results are new and stored in the variable `newResTable`.
- If you run the script a second time without modifying the files in between, there are no new results. The variable `newResTable` contains an empty table and an appropriate message is displayed.

If you modify files in between two runs, the variable `newResTable` contains only results related to the modifications.

```
proj = polyspace.Project;

% Specify sources and includes
sourceFile = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c');
includeFolder = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');

% Create results folder name based on time of analysis
runTime = datetime('now', 'Format', "d_MMM_y_H'h'_m'm");
resultsFolder = ['results_', char(runTime)];

% Configure analysis
proj.Configuration.Sources = {sourceFile};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.EnvironmentSettings.IncludeFolders = {includeFolder};
proj.Configuration.ResultsDir = fullfile(pwd, resultsFolder);

% Set up import from previous results if a previous result folder exists
if isfile('lastResultFolder.mat')
    load('lastResultFolder.mat', 'lastResultsFolder');
    proj.Configuration.ImportComments = fullfile(pwd, lastResultsFolder);
end
lastResultsFolder = resultsFolder;
save('lastResultFolder.mat', 'lastResultsFolder');

% Run analysis
bfStatus = run(proj, 'bugFinder');
```

```
% Read results
resObj = proj.Results;
resTable = getResults(resObj);
matches = (resTable.New == 'yes');
newResTable = resTable(matches,:);
if isempty(newResTable)
    disp('There are no new results.')
end
```

The key functions used in this example are:

- `polyspace.Project`: Run a Polyspace analysis and read the results to a table.
 - To specify a results folder, use the property `Configuration.ResultsDir`.
 - To specify a previous results folder to import results from, use the property `Configuration.ImportComments`.
- `datetime`: Read the current time, convert to an appropriate format, and append it to the results folder name.
- `load` and `save`: Load the previous results folder name from a MAT-file `lastResultFolder.mat` and save the current results folder name to the MAT-file for subsequent runs.

Review New Results and Unreviewed Results from Last Run

Instead of focusing on new results only, you can choose to focus on unreviewed results. Unreviewed results include new results and results from the last run that were not assigned a status in the Polyspace user interface.

To focus on unreviewed results, replace this section of the previous script:

```
% Read results
resObj = proj.Results;
resTable = getResults(resObj);
matches = (resTable.New == 'yes');
newResTable = resTable(matches,:);
if isempty(newResTable)
    disp('There are no new results.')
end
```

with this section:

```
% Read results
resObj = proj.Results;
resTable = getResults(resObj);
matches = (resTable.Status == 'Unreviewed');
unrevResTable = resTable(matches,:);
if isempty(unrevResTable)
    disp('There are no unreviewed results.')
end
```

See Also

`polyspace.Project` | `datetime` | `load` | `save`

More About

- “Run Polyspace Analysis by Using MATLAB Scripts” on page 5-9

Generate MATLAB Scripts from Polyspace User Interface

You can specify analysis options in the Polyspace user interface and later generate a MATLAB script for easier reuse of those options.

In the user interface, to determine which options to specify, you have tooltips, autocompletion of function names, context-sensitive help and so on. After you specify the options, you can generate a MATLAB script. For subsequent analyses, you can modify and run the script without opening the Polyspace user interface.

Prerequisites

Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

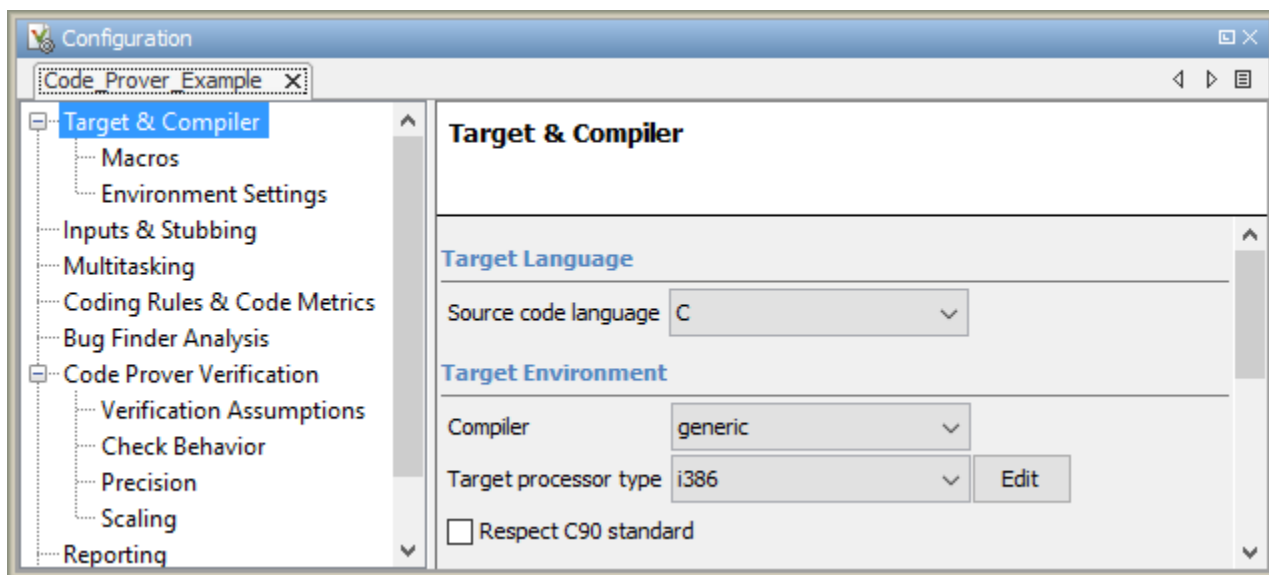
Create Scripts from Polyspace Projects

To start an analysis in the Polyspace user interface, create a project. In the project:

- You specify source and include folders during project creation.
- You specify analysis options such as compiler or multitasking in your project configuration. You also enable or disable checkers.

From this project, you can generate a script that contains your sources, includes and other analysis options. To begin, select **File > New Project**. For details, see “Add Source Files for Analysis in Polyspace Desktop User Interface” on page 2-2.

This example uses a sample project. To open the project, select **Help > Examples > Code_Prover_Example.psprj**. You see the options in the project configuration. For instance, on the **Target & Compiler** node, you see a generic compiler and an i386 processor.



- 1 Open MATLAB.

- 2 Create a `polyspace.Options` object from the sample Polyspace project.

```
projectFile = fullfile(polyspaceroot, 'polyspace', 'examples', 'cxx', ...
    'Code_Prover_Example', 'Code_Prover_Example.psprj');
opts = polyspace.loadProject(projectFile);
```

If a project has more than one module (with more than one configuration in each module), the options from the currently active configuration in the currently active module will be extracted in the options object. You cannot use the `loadProject` method on a project file that is created from a build command by using `polyspace-configure`.

- 3 Append the object to a MATLAB script.

```
filePath = opts.toScript('runPolyspace.m', 'append');
```

Open the script `runPolyspace.m`. You see the options that you specified from the user interface. For instance, you see the compiler and target processor.

```
opts.TargetCompiler.Compiler = 'generic';
opts.TargetCompiler.Target = 'i386';
```

Later, you can run the script to create a `polyspace.Options` object.

```
run(filePath);
```

The preceding example converts the sample project `Code_Prover_Example` directly to a script. When you open the sample project in the user interface, a copy is loaded into your Polyspace workspace. If you make changes to the sample project, the changes are made to the copied version. To see the changes in your MATLAB script, provide the copied project path to the `loadProject` method. To see the location of your workspace, select **Tools > Preferences** and view the **Project and Results Folder** tab.

See Also

Related Examples

- “Run Polyspace Analysis by Using MATLAB Scripts” on page 5-9

Troubleshoot Polyspace Analysis from MATLAB

When you run a Polyspace analysis on your C/C++ code, if one or more of your files fail to compile, the analysis continues with the remaining files. You can choose to stop the analysis on compilation errors.

```
proj = polyspace.Project;
proj.Configuration.EnvironmentSettings.StopWithCompileError = true;
```

However, it is more convenient to let the analysis complete and capture all compilation errors.

The compilation errors are displayed in the analysis log that appears on the MATLAB command window. The analysis log also contains the options used and the various stages of analysis. The lines that indicate errors begin with the `Error:` string. Find these lines and extract them to a log file for easier scanning. Produce a warning to indicate that compilation errors occurred.

Prerequisites

Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

Capture Polyspace Analysis Errors in Error Log

The function `runPolyspace` defined later captures the output from the command window using the `evalc` function and stores lines starting with `Error:` in a file `error.log`. You can call `runPolyspace` with paths to your source and include folders.

For instance, you can call the function with paths to demo source files in the subfolder `polyspace/examples/cxx/Bug_Finder_Example/sources` of the MATLAB installation folder.

```
sourcePath = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');
includePath = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');
[status, resultsSummary] = runPolyspace(sourcePath, includePath);
```

The function is defined as follows.

```
function [status, resultsSummary] = runPolyspace(sourcePath, libPath)
% runPolyspace takes two string arguments: source and include folder.
% The files in the source folder are analyzed for defects.
% If one or more files fail to compile, the errors are saved in a log.
% A warning on the screen indicates that compilation errors occurred.

    proj = polyspace.Project;

    % Specify sources
    proj.Configuration.Sources = {fullfile(sourcePath, '*')};

    % Specify compiler and paths to libraries
    proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
    proj.Configuration.EnvironmentSettings.IncludeFolders = {fullfile(libPath, '*')};

    % Run analysis
```

```

runMode = 'bugFinder';
[logFileContent,status] = evalc('run(proj, runMode)');

% Open file for writing errors
errorFile = fopen('error.log','wt+');

% Check log file for compilation errors
numErrors = 0;

log = strsplit(logFileContent,'\n');
errorLines = find(contains(log, {'Error:'}, 'IgnoreCase', true));
for ii=1:numel(errorLines)
    fprintf(errorFile, '%s\n', log{errorLines(ii)});
    numErrors = numErrors + 1;
end

if numErrors
    warning('%d compilation error(s). See error.log for details.', numErrors);
end

fclose(errorFile);

% Read results
resObj = proj.Results;
resultsSummary = getSummary(resObj, 'defects');
end

```

The analysis log is also captured in a file `Polyspace_R20##n_ProjectName_date-time.log`. Instead of capturing the output from the command window, you can search this file.

You can adapt this script for other purposes. For instance, you can capture warnings in addition to errors. The lines with warnings begin with `Warning:`. The warnings indicate situations where the analysis proceeds despite an issue. The analysis makes an assumption to work around the issue. If the assumption is incorrect, you can see errors later or in rare cases, incorrect analysis results.

See Also

`polyspace.Project`

Related Examples

- “Run Polyspace Analysis by Using MATLAB Scripts” on page 5-9
- “Troubleshoot Compilation Errors”

Run Polyspace Analysis in Simulink

Run Polyspace Analysis on Code Generated with Embedded Coder

If you generate code from a Simulink model by using Embedded Coder or TargetLink®, you can analyze the generated code for bugs or run-time errors with Polyspace from within the Simulink environment. You do not have to manually set up a Polyspace project.

This topic uses Embedded Coder for code generation. For analysis of TargetLink-generated code, see “Run Polyspace Analysis on Code Generated with TargetLink” on page 6-62.

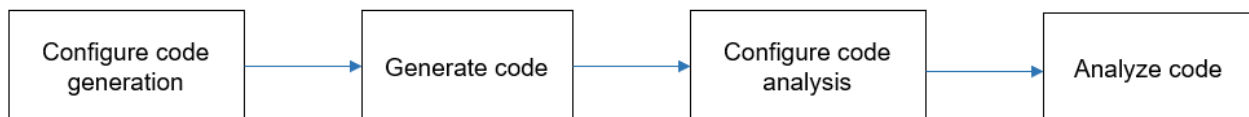
For a tutorial with a specific model, see “Run Polyspace Analysis on Code Generated from Simulink Model” on page 6-15.

You might want to analyze the generated code outside Simulink with other handwritten code. In this workflow, extract the Polyspace options and run the analysis, for instance, from the Windows Command Line. See “Run Polyspace Analysis on Generated Code by Using Packaged Options Files” on page 6-29. For older releases, Polyspace supports navigating from the generated code back to model. See “Navigate Back to Model” on page 6-71.

Prerequisites

Before you run Polyspace from Simulink, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

Generate and Analyze Code



Configure Code Generation and Generate Code

To configure code generation and generate code from a model, do *one of the following*:

- On the **Apps** tab, select **Embedded Coder**. Then, on the **C Code** tab, select **Quick Start**. Follow the on-screen instructions.
- On the **C Code** tab, click **Settings** and configure code generation through Simulink configuration parameters. The chief parameters to set are:
 - Type (Simulink): Select **Fixed-step**.
 - Solver (Simulink): Select **auto (Automatic solver selection)** or **Discrete (no continuous states)**.
 - System target file (Simulink Coder): Enter `ert.tlc` or `autosar.tlc`. If you derive target files from `ert.tlc`, you can also specify them.
 - Code-to-model (Embedded Coder): Select this option to enable links from code to model.

For the full list of parameters to set, see “Recommended Model Configuration Parameters for Polyspace Analysis” on page 6-51.

Alternatively, run the Code Generation Advisor with the objective **Polyspace** and see if the required parameters are already set. See “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder).

To generate code from the model, on the **C Code** tab, select **Generate Code**. You can follow the progress of code generation in the Diagnostic Viewer.

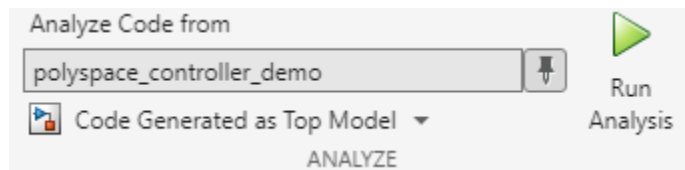
Configure Code Analysis

On the **Apps** tab, select **Polyspace Code Verifier**. On the **Polyspace** tab:

- 1 Select the product to run: **Bug Finder** or **Code Prover**. A Code Prover analysis detects run-time errors while a Bug Finder analysis detects coding defects and coding rule violations.
- 2 Select **Settings**. If needed, change default values of these options.
 - Settings from: Enable checking of MISRA™ coding rules in addition to the default checks specified in the project configuration. The default Bug Finder checks look for coding defects. The default Code Prover checks look for run-time errors.
 - “Input”, “Tunable parameters” and “Output”: Constrain inputs, tunable parameters, or outputs for a more precise Code Prover analysis.
 - “Output folder”: Specify a dedicated folder for results. The default analysis saves the results in a folder `results_modelName` in the current working folder.
 - “Open results automatically after verification”

Analyze Code

To analyze the code generated from the model, click anywhere on the canvas. The **Analyze Code from** field shows the model name. Select **Run Analysis**.



When using Embedded Coder, Polyspace checks for generated code when you click **Run Analysis**. If no generated code is present or if the model has changed since the last Polyspace analysis, Polyspace first launches the code generation process and then starts the analysis.

If the current model is referenced in another model and you want to verify the generated code in the context where the model is referenced, instead of **Code Generated as Top Model**, use **Code Generated as Model Reference**. In the latter case, Polyspace does not launch code generation automatically if there's no generated code. When analyzing **Code Generated as Model Reference**, generate code before running the Polyspace analysis.

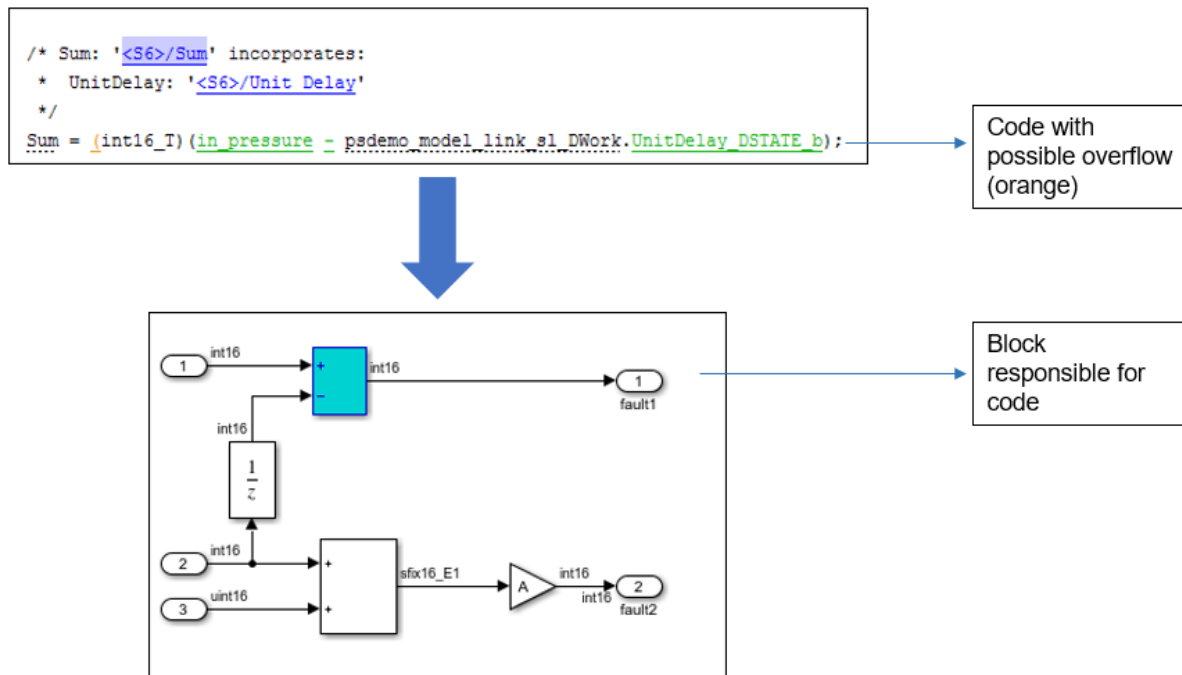
You can follow the progress of the analysis in the MATLAB Command Window.

The results open automatically unless explicitly disabled. By default, the results are saved in a folder `results_ModelName` in the current folder. Each new run overwrites previous results. You can

change the default folders or save the results to a Simulink project. To make these changes, on the **Polyspace** tab, select **Settings**.

If you have closed the results and want to open them later, on the **Polyspace** tab, select **Analysis Results**. To open a result prior to the last run, select **Open Earlier Results** and navigate to the folder containing the previous results.

Review Analysis Results



Review Results in Code

The results appear in the Polyspace user interface on the **Results List** pane. Click each result to see the source code on the **Source** pane and details on the **Result Details** pane. See also:

- “Interpret Bug Finder Results in Polyspace Desktop User Interface” on page 21-2
- “Bug Finder Defect Groups” on page 18-43
- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 22-2
- “Filter and Group Results in Polyspace Desktop User Interface” on page 23-2

Navigate from Code to Model

Links in code comments show blocks that generate the subsequent lines of code. To see the blocks in the model, click the block names in the links. If you encounter issues, see “Troubleshoot Navigation from Code to Model” on page 6-66.

Alternatively, you can right-click a variable name and select **Go to Model**. This option is not available for all variables. Only a subset of source code variables can be directly traced to a Simulink block.

The **Go to Model** options is available for such a variable. For more details on which variables in generated code can be traced to Simulink blocks, see “Trace Simulink Model Elements in Generated Code” (Embedded Coder).

Fix Issue

Investigate whether the issues in your code are related to design flaws in the model.

Design flaws in the model can lead to issues in the generated code. For instance:

- The generated code might be free of specific run-time errors only for a certain range of a block parameter. To fix this issue, you can change the storage class of that block parameter or use calibration data for the analysis by using the configuration parameter “Tunable parameters”.
- The generated code might be free of specific run-time errors only for a certain range of inputs. To determine this error-free range, you can specify a minimum and maximum value for the Inport block signals. The Polyspace analysis uses this constrained range. See “Work with Signal Ranges in Blocks” (Simulink).
- Certain transitions in Stateflow® charts can be unreachable.

You might integrate the generated code with handwritten code. A Polyspace analysis can detect coding defects and coding rule violations stemming from the integration. If you include any handwritten code in your Simulink model, you can analyze the included handwritten code in isolation. See:

- “Run Polyspace Analysis on Custom Code in C Function Block” on page 6-45
- “Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts” on page 6-37
- “Run Polyspace Analysis on S-Function Code” on page 6-35

Annotate Blocks to Justify Issues

You might want to justify some Polyspace results without modifying the code or the model. Annotate Simulink blocks either from the Polyspace user interface or the Simulink editor. See “Address Polyspace Results by Annotating Simulink Blocks” on page 6-6.

See Also

More About

- “Configure Polyspace Options in Simulink” on page 6-53

Address Polyspace Results by Annotating Simulink Blocks

When reviewing Polyspace results, you might want to address known Polyspace results by adding justifications. Annotate the relevant Simulink blocks with the justification in the Simulink Editor or the Polyspace User Interface. Polyspace supports annotating these results:

- Code Prover run-time error checks. See “Run-Time Checks” (Polyspace Code Prover).
- Bug Finder defects. See “Defects”.
- MISRA C:2004, MISRA AC AGC, and MISRA C:2012 coding rules. See “MISRA C:2004 Rules” and “MISRA C:2012 Directives and Rules”.
- MISRA C++:2008 coding rules. See “MISRA C++:2008 Rules”.
- CERT C and C++ rules. See “CERT C Rules and Recommendations” and “CERT C++ Rules”.
- AUTOSAR C++14 rules. See “AUTOSAR C++14 Rules”.
- ISO-17961 rules. See “ISO/IEC TS 17961 Rules”.
- Custom naming convention rules. See “Custom Coding Rules”.
- Software complexity guidelines. See “Guidelines”.

After you annotate a block, code operations generated from the block show results that are repopulated with your comments. If you annotate a subsystem block or a block that leads to a function call, code operations generated from the block do not show your comments in the analysis results. If the block is a Lookup Table, enable the `Stub lookup tables` instead of using annotations. See `Stub lookup tables`

In code generated by using Embedded Coder, there are known deviations from MISRA C:2012. See “Deviations Rationale for MISRA C:2012 Compliance” (Embedded Coder). Justify these known issues by annotating blocks.

Annotations in Simulink blocks or in generated code do not take the history of the analysis into account. If you update your model, the Polyspace results might change while the annotations do not. Updating the model might render the existing annotations outdated. Update your annotations when you update your model or generated code.

Annotate Blocks Through Polyspace User Interface

If you use Embedded Coder to generate code, you can annotate Simulink blocks directly through the Polyspace UI. Locate the issue that you want to annotate, and then enter review information by adding **Severity**, **Status**, and optional notes in the **Result Details** pane. For instance:

- Set the **Status** of the issue to `To Investigate`
- Set the **Comment** for the issue to `Might Impact "Module"`

In the source code, right-click the variable showing the issue (or another variable in the same expression) and from the context menu, select **Annotate Block**.

```

154 rtb_Switch = (uint16_T)psdemo_model_link_sl_B.Merge;
155
156 /* S-Function (Command_Strategy): '<S2>/Command_Strategy' */
157 psdemo_model_link_sl_B.Command_Strategy = command_strategy(rt_b_Switch,
158     in_battery_info);
159
160 /* Gain: '<S6>/Gain' incorporates:
161  * Sum: '<S6>/Sum1'
162  */
163 Gain = (int16_T)((int16_T)((in_rotation + in_battery_info) >> 1) * 24576) >>
164     10);
165
166 /* Sum: '<S6>/Sum' incorporates:
167  * UnitDelay: '<S6>/Unit_Delay'
168  */
169 Sum = (int16_T)(in_pressure - psdem
170
171 /* Switch: '<S7>/Switch' */
172 if (in_rate != 0) {
173     rt_b_Switch = in_battery_info;
174 } else {
175     rt_b_Switch = in_rate;
176 }
177
178 /* End of Switch: '<S7>/Switch' */
179
180 /* Sum: '<S10>/Sum2' incorporates:
181  * UnitDelay: '<S10>/Unit_Delay'
182  */
183 tmp = rt_b_Switch - psdemo_model link_sl_B.Command_Strategy;

```

The review information carries over to the Simulink Editor as block annotation where the annotated block is highlighted.



You can annotate a Simulink block multiple times. Subsequent annotations on a block are appended to previous annotations. These annotations cannot be seen in the Simulink Editor. When you analyze the generated code by using Polyspace, these annotations are displayed as review information in the **Result details** pane of the Polyspace UI.

The option **Annotate Block** is available for code elements that can be traced to a Simulink block. For more information, see “Trace Simulink Model Elements in Generated Code” (Embedded Coder).

Annotate Blocks in Simulink Editor

To annotate a block in the Simulink Editor, select the block and on the **Polyspace** tab, select **Add Annotation**. In the **Polyspace Annotation** window:

- Select the type of Polyspace result that you want to annotate from the drop-down list **Annotation Type**.
- If you want to annotate multiple results of the same type, enter a comma-separated list of result acronyms in the text box. See:
 - “Short Names of Bug Finder Defect Groups and Defect Checkers” on page 30-11
 - “Short Names of Code Prover Run-Time Checks” (Polyspace Code Prover)
- If you want to annotate only one result, select **Only 1 check**. The text box is converted into a dropdown list. Select the result that you want to annotate from this dropdown list.
- In the corresponding text boxes, enter the status, severity, and comment that you want to assign to the results.

In the **Polyspace Annotation** window, you can annotate a single type of Polyspace result at a time. To annotate multiple types of results, open the **Polyspace Annotation** window multiple times. Each time, add an annotation corresponding to one type of Polyspace result. The different annotations are appended to each other. These annotations cannot be seen in the Simulink Editor. When you analyze the generated code by using Polyspace, these annotations are displayed as review information in the **Result details** pane of the Polyspace UI.

Sometimes operations in the generated code cause orange checks in Code Prover. Suppose an operation potentially overflows. The generated code protects against the overflow by following the operation with a saturation. Polyspace still flags the possible overflow as an orange check. To justify these checks through code comments, specify the configuration parameter Operator annotations (Embedded Coder).

Limitations

When you copy an annotated block, and then use it in a different model or in a different position in the same model, the changed context can render the annotation incorrect:

- Polyspace does not allow annotation in blocks inside libraries and nonatomic subsystems because these blocks are reused in many different contexts. For instance, you cannot annotate a block inside a library block and justify results on all instances of the library block.
- Simulink does not retain Polyspace annotations in blocks that are copied to a different model or in a different position in the same model.

See Also

More About

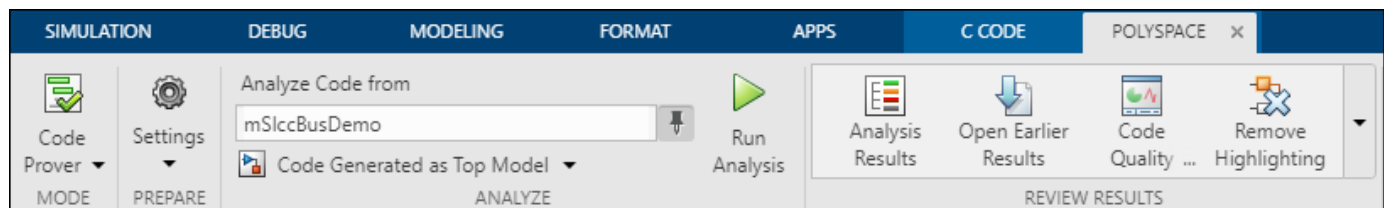
- “Configure Polyspace Options in Simulink” on page 6-53

Changes in Polyspace Analysis Workflows in Simulink in R2019b

In R2019b, a toolstrip with contextual buttons replaces the menus and toolbars in the Simulink Editor. The Simulink toolstrip includes contextual tabs, which appear only when you need them.

Code generation and verification tasks appear in separate tabs on the Simulink toolstrip.

- To generate code, open the **C Code** tab. To access this tab, on the **Apps** tab, select **Embedded Coder**.
- To analyze the generated code, open the **Polyspace** tab. To access this tab, on the **Apps** tab, select **Polyspace Code Verifier**.



Code Verification Workflow in a Nutshell

After code generation, on the **Polyspace** tab, use these steps to perform code verification:

- 1 Select product to run:

For instance, select **Bug Finder**.

- 2 Specify code analysis options:

Optionally, configure code analysis options. To configure the basic options related to the model, select **Settings > Polyspace Settings**. To configure advanced options related to the generated code, select **Settings > Project Settings**.

- 3 Specify which code to analyze:

Select whether to analyze the code generated for standalone use (typically, in the `modelName_ert_rtw` folder), the code generated for referencing in another context (typically, in the `s\prj` folder), or the custom code called from C Caller blocks or Stateflow charts.

- 4 Run analysis:

To start an analysis, select **Run Analysis**. The analysis runs on the model element selected, provided code has been generated earlier from the same element. The selected element appears in the **Analyze Code from** field. To select the entire model, click anywhere on the canvas outside a model element.

Locate Pre-R2019b Menu Items in Simulink Toolstrip

All menu items available earlier in the submenu **Code > Polyspace** now appear on the **Polyspace** tab.

Task	Before R2019b in Code > Polyspace menu	R2019b on Polyspace tab
Specify a Bug Finder analysis.	Select Options . Specify Bug Finder for the configuration parameter Product mode .	In the Mode group, select Bug Finder .
Run analysis on code generated from the model as standalone code. Typically, the analysis runs on the generated code in the <i>modelname_ert_rtw</i> folder.	Select Verify Code Generated for > Model .	Click anywhere on the canvas outside a model element. In the toolstrip, the Analyze Code from field displays the model name. Below the field, select Code Generated as Top Model . Then, select Run Analysis .
Run analysis on code generated from the model for reference in other models Typically, the analysis runs on the generated code in the <i>slprj</i> folder.	Select Verify Code Generated for > Referenced Model .	Click anywhere on the canvas outside a model element. In the toolstrip, the Analyze Code from field displays the model name. Below the field, select Code Generated as Model Reference . Then, select Run Analysis .
Configure basic analysis options related to the model.	Select Options .	Select Settings > Polyspace Settings .
Configure advanced analysis options related to the generated code.	Select Options . Click the Configure button next to the configuration parameter Project Configuration .	Select Settings > Project Settings .
Detach Polyspace options from model configuration for sharing with others who do not have Polyspace.	Select Remove Options from Current Configuration .	Select Settings > Remove Polyspace Configuration from Model .
Open results from the last Polyspace analysis on the model.	Select Open Results > For Generated Code or Open Results > For Generated Model Referenced Code .	Make sure that the Analyze Code from field states the model name (otherwise select anywhere on the canvas outside a model element). Below this field, select one of Code Generated as Top Model or Code Generated as Model Reference . Then, select Analysis Results .

Task	Before R2019b in Code > Polyspace menu	R2019b on Polyspace tab
<p>Open remote job monitor (if you are offloading the analysis to a server).</p>	<p>Select Open Job Monitor.</p> <p>For remote analysis, you must first set up communication with a server by using Polyspace preferences. See “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.</p>	<p>In the Review Results group, select Remote Job Monitor.</p> <p>For remote analysis, you must first set up communication with a server by using Polyspace preferences. See “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.</p>
<p>Open Polyspace Metrics or Polyspace Access web interface if you are using one of them to host Polyspace results.</p> <hr/> <p>Note Polyspace Metrics is removed in R2021b and later releases.</p>	<p>Select Open Metrics.</p> <p>For opening a web interface, you must first specify the hostname and port number used for the web server in Polyspace preferences.</p>	<p>In the Review Results group, select Code Quality Metrics (Polyspace Metrics) or Access (Polyspace Access).</p> <p>For opening a web interface, you must first specify the hostname and port number used for the web server in Polyspace preferences.</p>

See Also

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 6-2

Run Polyspace on Code Generated by Using Previous Releases of Simulink

You can use a more recent release of Polyspace without changing your Simulink release. See “Polyspace Support of MATLAB and Simulink from Different Releases” on page 6-68.

In such a cross-release configuration, use the function `pslinkrunCrossRelease` to run a Polyspace analysis on the code generated by using Embedded Coder. If you use Polyspace and Simulink from the same release, see “Run Polyspace Analysis on Code Generated from Simulink Model” on page 6-15.

Prerequisite

When starting a Polyspace analysis from a different release of MATLAB or Simulink:

- The Polyspace release must be more recent compared to your Simulink release.
- Your Simulink release must be R2020b or later.
- You must integrate Polyspace with Simulink. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

This cross-release configuration does not support analyzing the custom code in your Simulink model.

Run a Cross-Release Polyspace Analysis

To run a Polyspace analysis of code generated by using an earlier release of Simulink, generate code archive from the Simulink model and then call the function `pslinkrunCrossRelease`. Create and customize a `pslinkoptions` object to modify the model configuration. For a list of configuration options that you can modify, see `pslinkrunCrossRelease`. To apply Polyspace analysis options, use an options file.

- 1 Open the Simulink model `rtwdemo_roll` and configure the model for code generation. See “Recommended Model Configuration Parameters for Polyspace Analysis” on page 6-51.

```
% Load the model
model = 'rtwdemo_roll';
load_system(model);
% Configure the Solver
configSet = getActiveConfigSet(model);
set_param(configSet, 'Solver', 'FixedStepDiscrete');
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
```

- 2 The cross-release analysis requires packaging the generated code into a code archive. Set the option `PackageGeneratedCodeAndArtifacts` to true.

```
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true)
```

- 3 Create temporary folders for code generation and generate code.

```
[TEMPDIR, CGDIR] = rtwmoddir();
slbuild(model);
```

Alternatively, create a folder in a writable location and set your MATLAB directory to the created folder.

```
mkdir CodeGenFolder;
cd CodeGenFolder;
```


- 4 To specify the model configuration for the Polyspace analysis, use a `pslinkoptions` object. To run a Polyspace Bug Finder analysis, set `psOpt.VerificationMode` to `'BugFinder'`.

```
% Create a Polyspace options object from the model.
psOpts = pslinkoptions(model);
```

```
% Set properties that define the Polyspace analysis.
psOpts.VerificationMode = 'BugFinder';
```

Alternatively, set `psOpt.VerificationMode` to `'CodeProver'` to run a Polyspace Code Prover analysis.

- 5 To specify Polyspace analysis options, create an options file. An options file is a text file that contains Polyspace options in a flat list, one line for each option. For instance, to enable all checkers and CERT C coding rules, create a text file in the current folder containing the corresponding options.

```
% Create Options file
optFile = 'Options.txt';
fid = fopen(optFile,'wt');
option1 = '-checkers all';
option2 = '-cert-c all';
fprintf(fid, '%s\n%s', option1, option2);
fclose(fid);
```

See “Complete List of Polyspace Bug Finder Analysis Engine Options”.

- 6 Start a Polyspace analysis.

- To specify the model configurations for the Polyspace analysis run, set the object `psOpt` as the optional second argument in `pslinkrunCrossRelease`.
- Because the code is generated as standalone code, set the third argument `asModelRef` to `false`.
- To specify the Polyspace analysis options, specify the relative path to the created options file as the fourth argument.

```
% Locate options file in the current folder
optionsPath = fullfile(pwd,optFile);
% Run Polyspace analysis
[~,resultsFolder] = pslinkrunCrossRelease(model,psOpts,false,optionsPath);
bdclose(model);
```

Follow the progress of the analysis in the MATLAB Command Window.

Review Results

In a cross-release workflow, direct calls to functions such as `polyspaceBugFinder` or `polyspaceCodeProver` are not available. To open the results, use the function `pslinkfun`.

- 1 To open the results in the Polyspace User Interface, use the function `pslinkfun`. The character vector `resultsFolder` contains the full path to the results folder.

```
pslinkfun('openresults', '-resultsfolder',resultsFolder);
```

You can upload the results to Polyspace Access. See “Upload Results to Polyspace Access” on page 2-25.

- 2 Review the results, and fix or justify the identified issues. For more information, see “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 22-2.

See Also

`pslinkrunCrossRelease` | `polyspacePackNGo` | `slbuild` | `packNGo` | `pslinkfun`

More About

- “Run Polyspace Analysis on Generated Code by Using Packaged Options Files” on page 6-29
- “Run Polyspace Analysis by Using MATLAB Scripts” on page 5-9
- “Polyspace Support of MATLAB and Simulink from Different Releases” on page 6-68

Run Polyspace Analysis on Code Generated from Simulink Model

This tutorial shows how to run a Polyspace analysis on C/C++ code generated from a Simulink model. You can also analyze C/C++ code generated from a subsystem. For the complete workflow, see “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 6-2.

Prerequisites

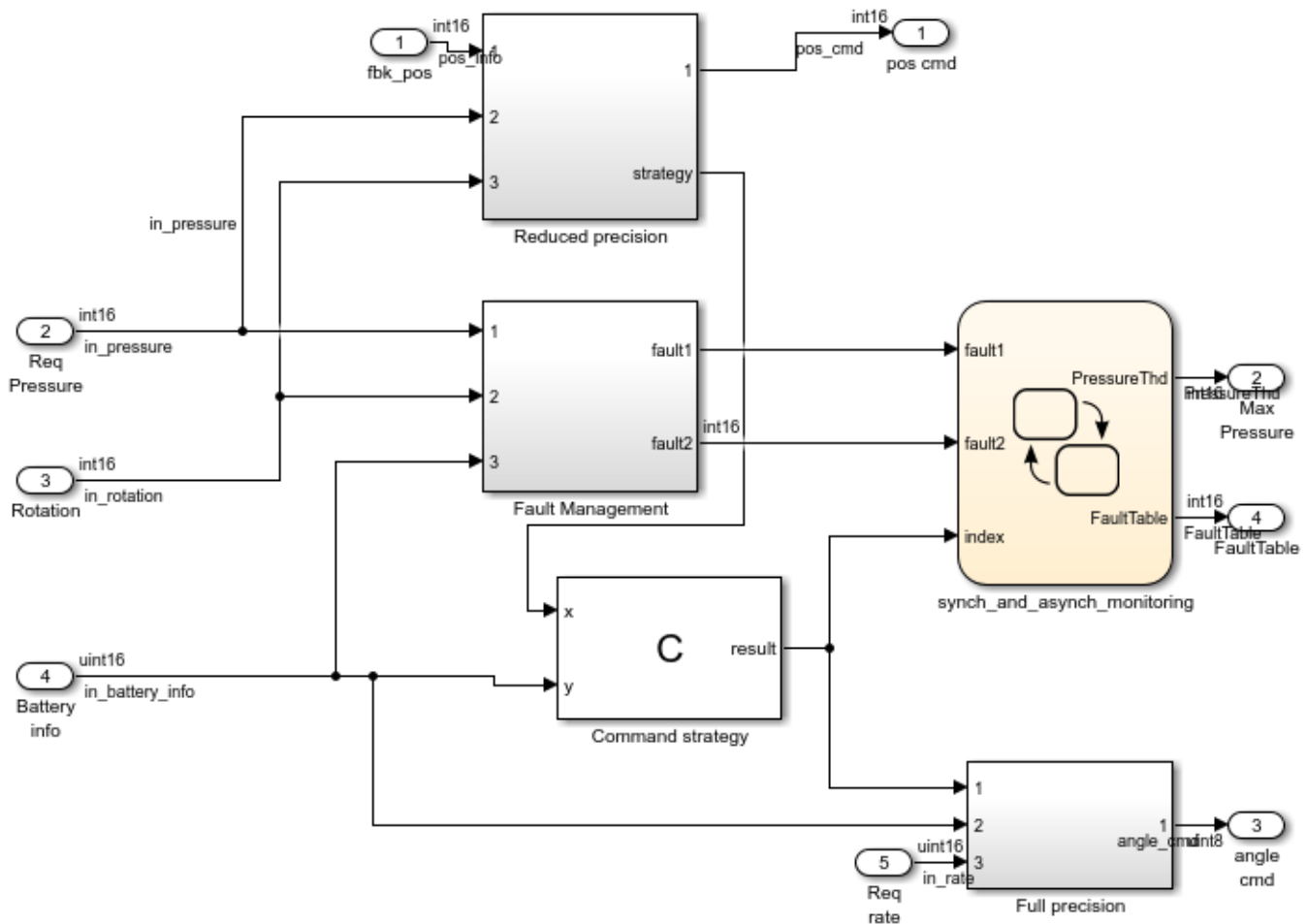
Before you run Polyspace from Simulink, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

To open the model used in this example, in the MATLAB Command Window, run:

```
openExample('polyspace_code_prover/OpenSimulinkModelForPolyspaceAnalysisExample')
```

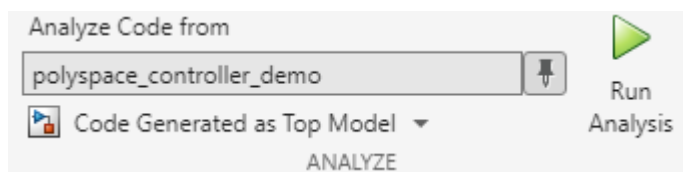
Open Simulink Model for Polyspace Analysis

Open the model `polyspace_controller_demo`.



Check for Run-Time Errors in Generated Code

- 1 On the **Apps** tab, select **Polyspace Code Verifier**. The **Polyspace** tab opens.
- 2 On the **Polyspace** tab, select **Code Prover** in the **Mode** section.
- 3 Locate the **Analyze** section and select **Code Generated as Top model** from the drop-down list.
- 4 Click **Run Analysis**. Polyspace checks if the model has been changed since the last code generation. If the generated code is up-to-date, Polyspace starts the analysis. If the generated code is not up-to-date or if there is no generated code, Polyspace generates the code first and then starts the analysis.



Alternatively, to start the analysis from the MATLAB Command Window, enter:

```

% Load model
load_system('polyspace_controller_demo');
% Generate code
slbuild('polyspace_controller_demo');
% Create Polyspace options object
mlopts = pslinkoptions('polyspace_controller_demo');
% Specify result folder
mlopts.ResultDir = '\cp_result';
% Set analysis to Code Prover mode
mlopts.VerificationMode = 'CodeProver';
% Run analysis
pslinkrun('polyspace_controller_demo', mlopts);

```

For more information about running Polyspace analysis in the MATLAB Command Window, See `pslinkoptions` and `pslinkrun`.

Review Analysis Results

After the analysis completes, the analysis results appear in the Polyspace user interface. The results consist of color coded checks:

- **Green** (✓): The check appear on proven code that does not fail for the data constraints provided. For instance, a division operation does not cause a **Division by Zero** error.
- **Red** (●): The check appear on a verified error that always fails for the set of data constraints provided. For instance, a division operation always causes a **Division by Zero** error.
- **Orange** (?): The check indicates a possible error in unproven code that can fail for certain values of the data constraints provided. For instance, a division operation sometimes causes a **Division by Zero** error.
- **Gray** (✕): The check indicates a code operation that cannot be reached for the data constraints provided.

Review each result in detail. In your Code Prover results:

- 1 On the **Results List** pane, locate the red **Out of bounds array index** check. Click the red check (●).
- 2 On the **Source** pane, place your cursor on the red check on the `[]` operator to view the tooltip. It states the array size and possible values of the array index. The **Result Details** pane also provides this information.

Both red checks occur in the handwritten C code in the C Function block `Command_Strategy`.

Trace and Fix Issues in the Model

Issues reported by Polyspace on generated code might be caused by issues in the model. Trace an issue back to your model to investigate the root cause. Issues in code might occur due to a design issue such as:

- Faulty scaling, unknown calibrations, and untested data ranges coming out of a subsystem into an arithmetic block.
- Saturations leading to unexpected data flow inside the generated code.

- Faulty programming in custom code blocks such as the C Function and Stateflow blocks.

To fix design issues in the example model, identify the root cause of run-time errors reported by Polyspace:

Illegally dereferenced pointer

- 1 On the **Results List** pane, select the **Illegally dereferenced pointer** check.
- 2 On the **Source** pane, click the link **<Root>/Command strategy** in the comments above the error.

```
/* CFunction: '<Root>/Command strategy' incorporates:
 * DataTypeConversion: '<S3>/Cast4'
 * Inport: '<Root>/Battery info'
 */
//...
p = &array[0];
for (i = 0; i < 100; i++) {
    *p = 0;
    p = &p[1];
}
rtb_x = (int16_T)((uint16_T)rtb_y1 - in_battery_info);
if (rtb_x < 3) {
    rtb_x = (int16_T)(*p + 5);
}
//...
```

The Simulink Editor highlights the C Function block from which this error arises. In this block, the pointer `p` is incremented 100 times, pointing `*p` outside the bound of array. The dereferencing operation in `rtb_x = (int16_T)(*p + 5);` then causes a red **Illegally dereferenced pointer** check.

One solution for this error is to point `*p` to a valid memory location after the for loop in the C Function block:

```
// After the for loop, point p to a valid memory location
p = &(array[50]);
// ...
tmp = *p + 5;
```

Out of bounds array index

- 1 On the **Results List** pane, select the **Out of bounds array index** check.
- 2 On the **Source** pane, click the link **<Root>/Command strategy** in the comments above the error.

```
/* CFunction: '<Root>/Command strategy' incorporates:
 * DataTypeConversion: '<S3>/Cast4'
 * Inport: '<Root>/Battery info'
 */
//...
for (i = 0; i < 100; i++) {
    *p = 0;
    p = &p[1];
}
//...
if ((rtb_x > 92) && (rtb_x < 110)) {
    if (another_array[(rtb_x - i) + 9] != 0) {
```

```

    rtb_x = 92;
  } else {
    rtb_x = 91;
  }
}

```

The Simulink Editor highlights the C Function block from which this error arises. In this block, the value of `i` is set to 100 after the first for loop. The statement `if ((rtb_x > 92) && (rtb_x < 110))` limits the possible value of `rtb_x` to 93..109. In the statement `another_array[(rtb_x - i) + 9] != 0`, the possible indices for `another_array` range from $93+9-100 = 2$ to $109+9-100 = 18$. Because the array `another_array` has only two elements, the array access in `another_array[(rtb_x - i) + 9]` results in a red **Out of bounds array index** run-time check.

One solution for this error is to modify the prevailing conditions on `rtb_x` so that the expression `[(rtb_x - i) + 9]` evaluates to 0 or 1.

```

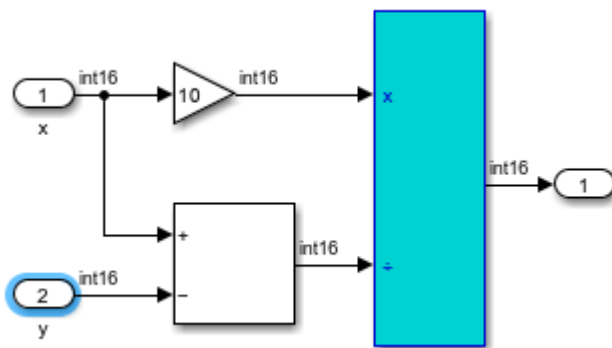
if ((rtb_x > 91) && (rtb_x < 93)) {
  if (another_array[(rtb_x - i) + 9] != 0) {
    rtb_x = 92;
  } else {
    rtb_x = 91;
  }
}
}

```

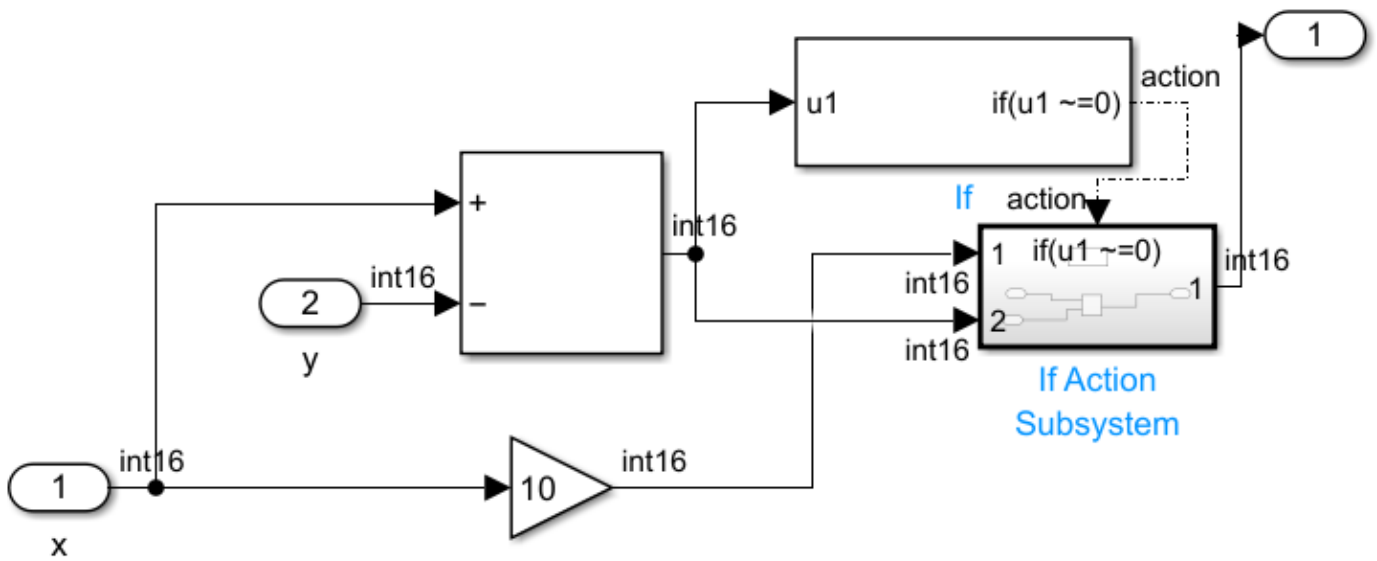
Orange checks

The orange checks represent run-time errors that might occur in specific code execution path. Review the orange checks and triage the source of these potential issues. For instance:

- **Division by zero** — This orange check is reported twice. One of these checks is reported in the statement `rtb_y1 = (int16_T)((int16_T)(10 * 10) / (int16_T)(10 - rtb_x))`. To trace the cause of this possible error, click the comment `<S6>/Divide`. The Simulink Editor highlights the Divide block. In the execution paths where the `÷` input equals zero, the division operation results in a **Division by zero** error.

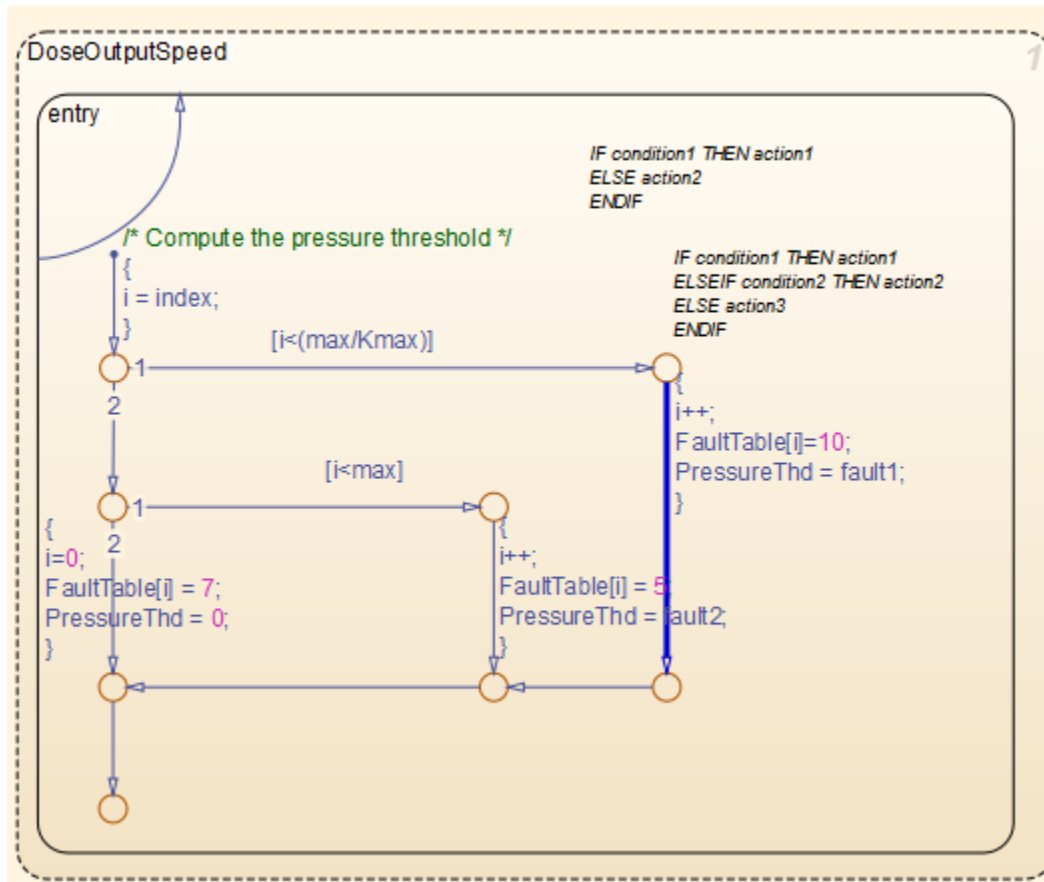


To resolve this error, check that the `÷` input is not zero. For instance, use the If block and put the Divide block in an If Action Subsystem.



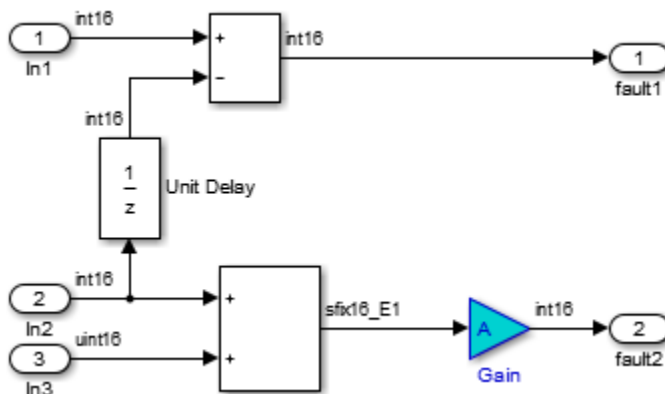
The other **Division by zero** checks can be resolved using similar techniques to check for a zero denominator.

- **Out of bound array index:** This orange check is reported on the statement `polyspace_controller_demo_Y.FaultTable[*i] = 10;`. To trace the root cause of this potential error, click the link **S4:76** in the comments above the orange error. The Simulink Editor highlights the Stateflow chart `synch_and_async_monitoring`. Trace the error to the input variable index of the Stateflow chart.



One solution to avoid this check is to constrain the input variable `index`. Use a Saturation block before the Stateflow chart to limit the value of `index` from zero to 100.

- Overflow:** Polyspace reports several orange **Overflow** checks. Resolve these checks by constraining the inputs. For instance, consider the orange **Overflow** check in the statement `rtb_k = (int16_T)((int16_T)((in_rotation + in_battery_info) >> 1) * 24576) >> 10`. To trace the check back to the model, click the link **S1/Gain** in the comments above the orange check. The Simulink Editor highlights the Gain block in the Fault Management subsystem.



One solution to avoid the orange **Overflow** checkk is to constrain the value of the signal `in_battery_info` that is fed to the Sum block. For instance:

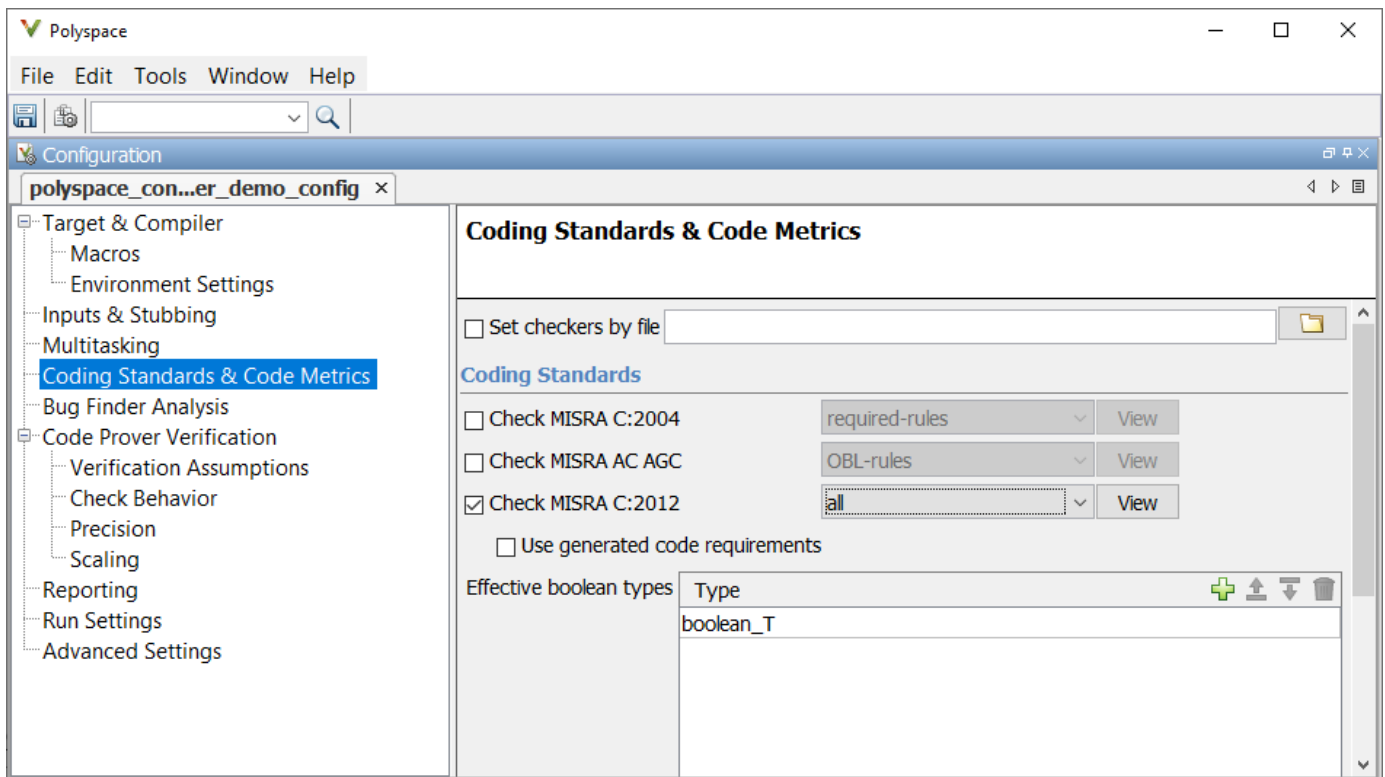
- 1 Double-click the Inport block `Battery_info` that provides the input signal `in_battery_info` to the model.
- 2 On the **Signal Attributes** tab, change the **Maximum** value of the signal to a lower value, such as 500.

Use this technique to address similar orange **Overflow** checks.

Check for Coding Rule Violations

To check for coding rule violations, start a Polyspace Bug Finder analysis.

- 1 On the **Polyspace** tab, select **Settings > Project Settings** and enable the MISRA C:2012 coding standard in the **Coding Standards & Code Metrics** node. Save the configuration and close the window.



- 2 In the **Mode** section, select **Bug Finder**.
- 3 Rerun the analysis.

Alternatively, in the MATLAB Command Window, enter:

```
% Enable checking for MISRA C:2012 violations
mlopts.VerificationSettings = 'PrjConfigAndMisraC2012';
% Specify separate folder for Bug Finder analysis
mlopts.ResultDir = '\bf_result';
% Set analysis to Bug Finder mode
```

```
mlopts.VerificationMode = 'BugFinder';
% Run analysis
pslinkrun('polyspace_controller_demo', mlopts);
```

After the analysis completes, the Polyspace UI opens containing a list of MISRA C:2012 rule violations.

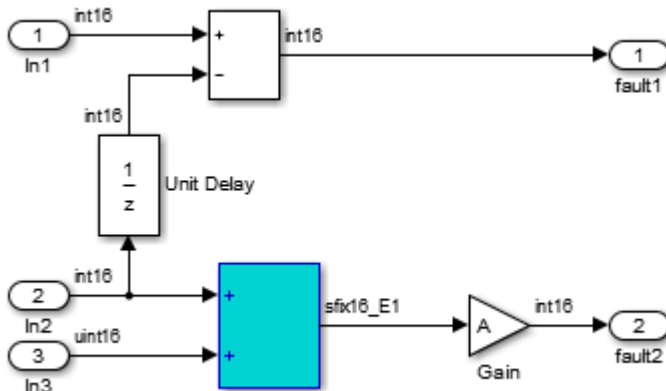
Annotate Blocks to Justify Results

To justify Polyspace results, add annotations to your blocks. During code analysis, Polyspace populates the results with your justification. Once you justify a result, you do not have to review it again in subsequent analyses.

- 1 On the **Results List** pane, from the list in the upper-left corner, select **File**.
- 2 In the file `polyspace_controller_demo.c`, in the function `polyspace_controller_demo_step()`, select the violation of MISRA C:2012 rule 10.4. The **Source** pane shows that an addition operation violates the rule.
- 3 On the **Source** pane, click the link **S1/Sum1** in the comments above the addition operation.

```
/* Gain: '<S1>/Gain' incorporates:
 * Inport: '<Root>/Battery Info'
 * Inport: '<Root>/Rotation'
 * Sum: '<S1>/Sum1'
 */
rtb_k = (int16_T)((((int16_T)((in_rotation + in_battery_info) >> 1) * 24576) >>
                10);
```

The rule violation occurs in a Sum block.



- 4 To annotate this block and justify the rule violation:
 - a Select the block. On the **Polyspace** tab, select **Add Annotation**.
 - b Select **MISRA-C-2012** for **Annotation type** and enter information about the rule violation. Set the **Status** to **No action planned** and the **Severity** to **Unset**.
 - c Click **Apply** or **OK**. The words **Polyspace annotation** appear below the block, indicating that the block contains a code annotation.
- 5 Regenerate code and rerun the analysis. The **Severity** and **Status** columns on the **Results List** pane are now prepopulated with your annotations.

See Also

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 6-2
- “Run Polyspace on Code Generated by Using Previous Releases of Simulink” on page 6-12

Fix Model Design Issues Found as Run-time Errors and Coding Rule Violations in Generated Code

After testing your Simulink model for standards compliance and design errors, you can generate code from the model. Before deployment, you can perform a final layer of error checking on the generated code by using Polyspace. The checks detect issues such as dead logic or incorrect code generation options that can remain despite tests on the model.

Prerequisites

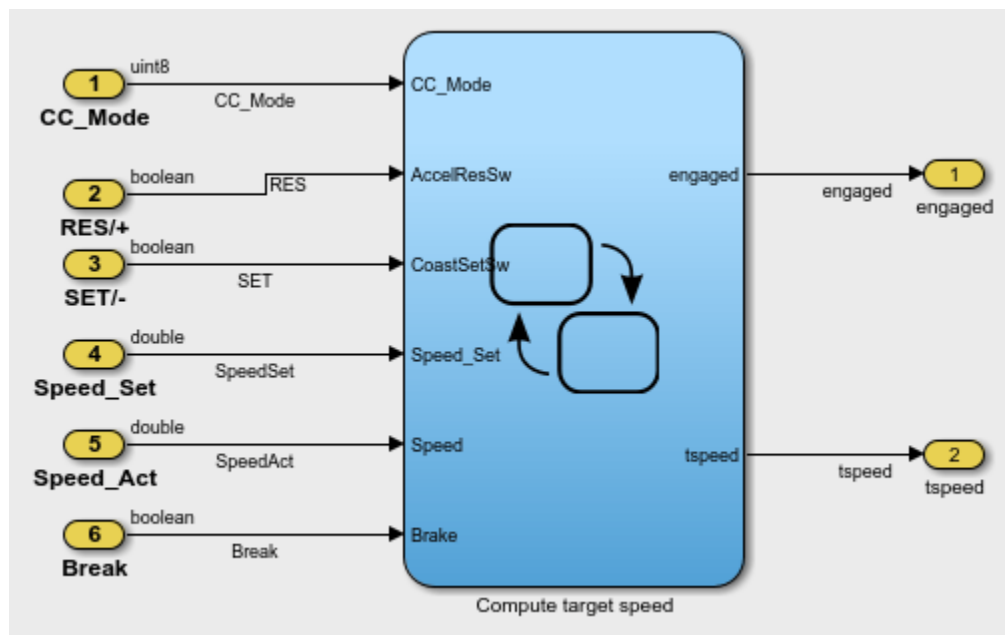
Before you run Polyspace from Simulink, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

To open the model used in this example, at the MATLAB Command Window, run:

```
openExample('polyspace_code_prover/FixIssuesInGeneratedCodeFoundWithPolyspaceCodeProverExample')
```

Open Model

The model `CruiseControl_RP` contains a Stateflow chart with design issues. The issues translate to possible run-time errors or unreachable branches in the generated code.



Detect and Fix Run-Time Errors

Detect Run-Time Errors

On the **Polyspace** tab, click anywhere on the canvas. The **Analyze Code from** field shows the model name. If you use Embedded Coder, then click **Run Analysis**. If you use other code generating tools, manually generate the code before starting a Polyspace analysis.

For more information, see “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 6-2.

After the analysis is complete, the Code Prover results open in the Polyspace user interface. The results contain gray checks (unreachable code) and orange checks (potential run-time errors).

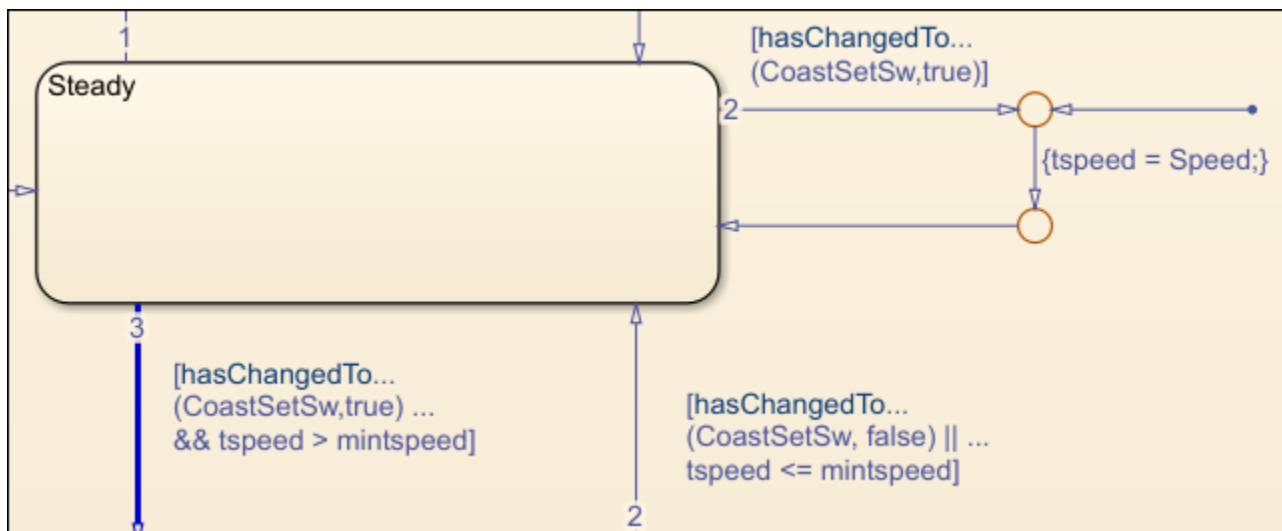
Fix Gray Checks

Select one of the two **Unreachable code** checks. Review the code that is unreachable.

```
if ((CoastSetSw_prev != CruiseControl_RP_DW.CoastSetSw_start) &&
    CruiseControl_RP_DW.CoastSetSw_start &&
    (CruiseControl_RP_Y.tspeed > (real_T)mintspeed)) {
/* Transition: '<S1>:74' */
CruiseControl_RP_DW.is_ON = CruiseControl_RP_IN_Coast;
CruiseControl_RP_DW.temporalCounter_i1 = 0U;

/* Entry 'Coast': '<S1>:73' */
CruiseControl_RP_Y.tspeed -= (real_T)incdec;
}
```

Click the Transition: '<S1>:74' link in the if block. The transition is highlighted in the model.



Note the design flaw. The condition for outgoing transition 3 cannot be true without the condition for outgoing transition 2 also being true. Therefore, transition 3, which executes later, is never reached. This design flaw in the chart translates to the unreachable `if` block in the generated code.

One possible solution of the issue is to switch the execution order of transitions 2 and 3. To begin, right-click transition 3.

After switching the execution order, regenerate and reanalyze the code. You no longer see the gray **Unreachable code** checks.

Fix Orange Checks

Select one of the two **Division by zero** checks. Review the code.

```
if (CruiseControl_RP_DW.temporalCounter_i1 >= (uint32_T)(incdec /
    holdrate * 10.0F))
```

Place your cursor on the variable `holdrate`. You see that it is a global variable whose value can be zero.

The fact that `holdrate` is a global variable hints that it could be defined outside the model. Open the Model Explorer window. In the model hierarchy, choose the base workspace. Find `holdrate` in the list of parameters. You see that `holdrate` has a value 5, but the value can range from 0 to 10. The Code Prover analysis uses this range and detects a division by zero.

You can modify either the generated code or the analysis configuration:

- Modify code:

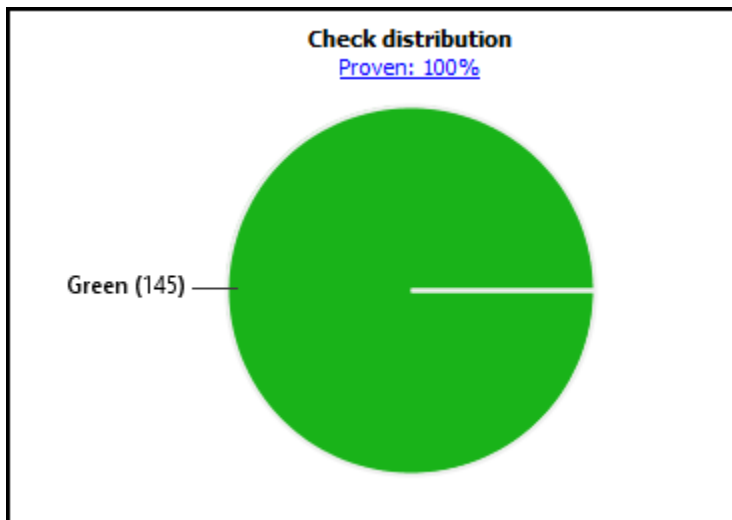
In the Model Explorer window, change the storage class of `holdrate` from `Global` to `Define`. The generated code defines a preprocessor directive stating that `holdrate` has the value of 5.

```
#define holdrate 5
```

- Modify analysis configuration:

On the **Polyspace** tab, select **Settings**. Modify the option **Tunable parameters** to use the calibration data. The Code Prover analysis uses the value 5 for `holdrate` instead of a different value in the range 0 to 10.

If you regenerate and reanalyze the code, you no longer see the orange **Division by zero** checks or any other orange checks that have the same root cause. The **Dashboard** pane shows that all checks are green.



Detect and Fix Coding Rule Violations

Detect MISRA C:2012 Violations

To detect MISRA C violations:

- 1 In the **Mode** section of the **Polyspace** tab, select **Bug Finder**.

- 2 Select **Settings** to open the Simulink Configuration Parameters window. In the **Settings from** menu, select Project configuration and MISRA C 2012 checking.
- 3 Start the analysis by clicking **Run Analysis**.

Fix MISRA C:2012 Violations

After running the Bug Finder analysis, Polyspace reports the violations of MISRA C:2012 in the generated code. To fix some of these violations, you might need to modify the model. Consider the violation of rule 3.1:

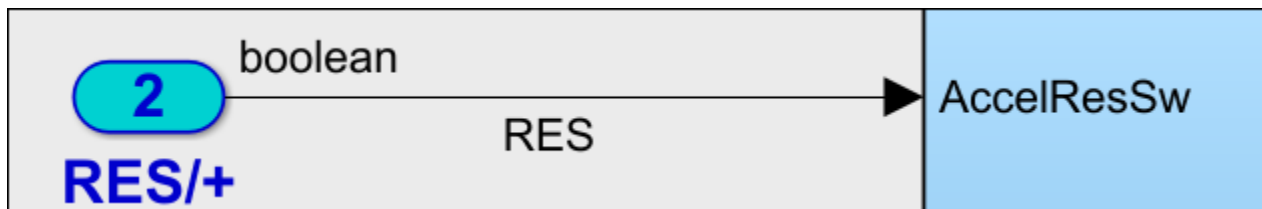
The character sequences `/*` and `//` shall not be used within a comment.

Two of the violations is located in this code on the declarations of RES and SET:

```
typedef struct {
    uint8_T CC_Mode;           /* '<Root>/CC_Mode' */
    boolean_T RES;            /* '<Root>/RES//+' */
    boolean_T SET;            /* '<Root>/SET//-' */
    real_T SpeedSet;          /* '<Root>/Speed_Set' */
    real_T SpeedAct;          /* '<Root>/Speed_Act' */
    boolean_T Break;          /* '<Root>/Break' */
} ExtU_CruiseControl_RP_T;
```

In these statements, you see two instances of `//` in the code comments in the structure definition.

To navigate to the corresponding location in the model, click '`<Root>/RES//+`' in the code comment. You see that the comment comes from the input variable RES/+, which contains the / character.



Rename the variables RES/+ and SET/- so that they do not use the / character. When you reanalyze the code, you no longer see violations of rule 3.1.

See Also

Related Examples

- “Run Polyspace Analysis on Code Generated from Simulink Model” on page 6-15

Run Polyspace Analysis on Generated Code by Using Packaged Options Files

When you start a Polyspace analysis directly from the Simulink toolstrip, the analysis takes the model-specific context, such as design ranges, into consideration. When running a Polyspace analysis from outside Simulink, you must specify the model-specific information by using options files. Instead of authoring these options files, use the options files generated and packaged by the function `polyspacePackNGo`.

Preserving the Simulink model context information when running a Polyspace analysis from outside Simulink can be useful in various situations. For instance:

- **Distributed workflow:** A Simulink user generates code from a model and sends the code to another development environment. In this environment, a Polyspace user, who might not have Simulink, runs a separate analysis of the generated code. By using the packaged options files, the design ranges and other model-specific information is preserved in the Polyspace analysis.
- **Analysis options not available in Simulink:** Some Polyspace analysis options are available only when the Polyspace analysis is run separately from Simulink. Use packaged options files to run a separate Polyspace analysis while preserving the model-specific information. For instance, analyze concurrent threads in generated code by running a Polyspace analysis in the generated code by using the packaged options files.

You must have Simulink to run the function `polyspacePackNGo`. You do not need Polyspace to generate the options files from a Simulink model. The `polyspacePackNGo` function supports code generated by Embedded Coder and TargetLink. For a tutorial on using `polyspacePackNGo`, see “Analyze Code Generated as Standalone Code in a Distributed Workflow” (Simulink).

Generate and Package Polyspace Options Files

To generate and package Polyspace options file for analyzing code generated from a Simulink model, use `polyspacePackNGo`.

- 1 In the Simulink Editor, open the Configuration Parameters dialog box and configure the model for code generation.
- 2 To configure the model for compatibility with Polyspace, select `ert.tlc` as the **System target file**
- 3 To enable generating a code archive, select the option **Package code and artifacts**. Optionally, provide a name for the options package in the field **Zip file name**. If your code contains a custom code block, select **Use the same custom code settings as Simulation target** in the **Code Generation > Custom Code** pane.

Alternatively, in the MATLAB Command Window, enter:

```
% Configure the Simulink model mdlName for code generation
configSet = getActiveConfigSet(mdlName);
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true);
set_param(configSet, 'PackageName', 'CodeArchive.zip');
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
set_param(configSet, 'RTWUseSimCustomCode', 'on');
```

- 4 Generate the code archive.

- To generate an archive of standalone generated code from the top model, use the function `slbuild`.
 - To generate code as a model reference, use the function `slbuild`. After generating code as model reference, create the code archive by using the function `packNGo`.
 - Alternatively, you can use `TargetLink` to generate the code. Create the code archive by archiving the generated code into a zip file.
- 5 To generate and package the Polyspace option files, in the MATLAB Command Window ,use the `polyspacePackNGo` function :

```
zipFile = polyspacePackNGo mdlName);
```

See “Generate and Package Polyspace Options Files”.

If you use `TargetLink` to generate code, then use the `TargetLink` subsystem name as the input argument to `polyspacepacknGo`.

- 6 Optionally, you can use a `pslinkoptions` object as a second argument to modify the default model configuration for the Polyspace analysis. Create a `pslinkoptions` object, modify model configurations and specify the object when creating the archive:

```
psOpt = pslinkoptions(mdlName);
psOpt.InputRangeMode = 'FullRange';
psOpt.ParamRangeMode = 'DesignMinMax';
zipFile = polyspacePackNGo(mdlName,psOpt);
```

See “Package Polyspace Options Files That Have Specific Polyspace Analysis Options”.

- 7 Use the optional third argument to specify whether to generate and package Polyspace options files for code generated as a model reference. Suppose you generated code as a model reference by using the `slbuild` function. To generate and package Polyspace options for the code, at the MATLAB Command Window, enter:

```
zipFile = polyspacePackNGo(mdlName,[],true);
```

See “Package Polyspace Options Files for Code Generated as a Model Reference”.

The function `polyspacepackNGo` returns the full path to the archive containing the options files. The files are located in the `polyspace` folder within the archived folder hierarchy. The content of the `polyspace` folder depends on the inputs of `polyspacePackNGo` function.

- If you do not specify the optional second and third arguments, then the folder `polyspace` contains these options files in a flat hierarchy:
 - `optionsFile.txt`: This file specifies the source files, the include files, data range specifications, and analysis options required for analyzing the generated code by using Polyspace. If your code contains custom C code, then this file specifies the relative paths of the custom source and header files.
 - `modelName_drs.xml`: This file specifies the design range specification of the model.
 - `linksData.xml`: This file links the generated code to the components of the model.
- If you specify `psOpts.ModelbyModelRef = true`, then corresponding options files are generated for all referenced models. These options files are stored in separate folders named `polyspace_<referenced model name>` within the code archive. The folder `polyspace` contains the options files for the top model.

Run Polyspace Analysis by Using the Packaged Options Files

Once the code archive and the Polyspace option files are generated, you can use the archive to run a Polyspace analysis on the generated code in a different development environment without Simulink.

- 1 Unzip the code archive and locate the `polyspace` folder.
- 2 On a Windows or Linux command line, run: `productname -options-file optionsFile.txt -results-dir resultdir`.
 - `productname` corresponds to one of: `polyspace-bug-finder`, `polyspace-code-prover`, `polyspace-bug-finder-server`, or `polyspace-code-prover-server`.
 - `resultdir` corresponds to the location of the Polyspace results. This argument is optional.

To link the generated code with the Simulink model, the file `linksData.xml` is required. In case the file `linksData.xml` is not generated in the options file archive, use the option **Code Generator Support** in Polyspace desktop User Interface to specify which comments in the code act as links to the Simulink model. In the Polyspace desktop User Interface, select **Tools > Preferences** and locate the **Miscellaneous** tab. From the context menu **Code comments that act as code-to-model-link**, select the code generator that you used. If you select **User defined**, then specify the comments that act as a code-to-model link by specifying their prefix in the field **Comments beginning with**. For instance, if you specify the prefix as `//Link_to_model`, then Polyspace interprets comments starting with `//Link_to_model` as links to model.

If you are using Polyspace Access to view the results, upload the file `linksData.xml` in the same folder as your Polyspace results. You cannot link the code with Simulink model if you do not have the file `linksData.xml` or if you upload it outside the Polyspace result folder.

- 3 To review the result, upload it to Polyspace Access and view the results in a web browser. Alternatively, view the result by using the user interface of the Polyspace desktop products.

See Also

`polyspacePackNGo` | `polyspace.Project` | `slbuild` | `packNGo`

More About

- “Analyze Code Generated as Standalone Code in a Distributed Workflow” (Simulink)
- “Run Polyspace Analysis by Using MATLAB Scripts” on page 5-9
- “Integrate Polyspace Server Products with MATLAB” on page 8-33

Run Polyspace Analysis on Custom Code in Simulink Models

If you implement algorithms in your Simulink model by using custom C/C++ code, you can analyze the custom code directly from the Simulink toolstrip without manually setting up a Polyspace project. The behavior of the custom code in your model depends on the model context, such as number and nature of input and design range specification. When you run Polyspace analysis from MATLAB or Simulink, the analysis takes the model context into account. When running a Polyspace analysis of the custom code outside of MATLAB/Simulink, specify the model context manually, for instance, by using options files.

A Polyspace analysis of the custom code has different goals and configurations compared to a Polyspace analysis of the generated code:

Generated Code Analysis	Custom Code Analysis
Analyzes the code in a C Caller, C Function, or S-Function block in isolation.	Analyzes the code generated from the entire model.
Detects issues in the custom code that can cause bugs or run-time errors in a Simulink simulation.	Detects issues in the total generated code that might cause bugs or run-time errors when deployed to an embedded system.
The target settings for Polyspace is compatible with a Simulink simulation.	The Target processor type settings for Polyspace is the same as the Hardware Implementation settings specified in the Configuration Parameters dialog box in Simulink.

Prerequisite

Before you run Polyspace with Simulink, link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2 or “Integrate Polyspace Server Products with MATLAB” on page 8-33.

Analyze Custom Code

You can implement custom algorithm by using different Simulink custom code blocks, such as:

- C Function: See “Integrate External C/C++ Code into Simulink Using C Function Blocks” (Simulink)
- C Caller: See “Integrate C Code Using C Caller Blocks” (Simulink)
- S-Function: See “Implement C/C++ S-Functions” (Simulink)

These blocks have different functionalities. See “Comparison of Custom Block Functionality” (Simulink).

Specify Configuration

Before running Polyspace on a Simulink model, configure the Simulink model to be compatible with Polyspace.

To analyze custom code in Polyspace, select **Import custom code** in the Configuration Parameters dialog box, on the **Simulation Target** pane.

If the included custom code does not compile, the Polyspace analysis might fail. Before starting the Polyspace analysis, include the appropriate header files and check the custom code for compilation issues. The C function block does not support including header files. For this block, specify the include statements in the **Simulation Target** pane. For the code included in C Caller and S Function blocks, specify the include statements in the source file. Polyspace has stricter code and compilation requirements than Simulink and your custom code might fail Polyspace compilation even though your model simulation produces correct results.

Start Polyspace Analysis

Start the Polyspace analysis of custom code in your model in the Simulink Editor or in the MATLAB Command Window.

- For more information about running a Polyspace analysis on custom code in a S function block, see “Run Polyspace Analysis on S-Function Code” on page 6-35.
- For more information about running a Polyspace analysis on custom code in a C Caller block, see “Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts” on page 6-37.
- For more information about running a Polyspace analysis on custom code in a C function block, see “Run Polyspace Analysis on Custom Code in C Function Block” on page 6-45.

Once the analysis starts, Polyspace extracts the custom code from the model. To preserve the correct design range specification and nature of the inputs, Polyspace assumes each instance of a custom code block in a top model has a unique model context and treats the blocks as unique. When a model containing a custom code block is referenced multiple times in another top model, the model context of the custom code blocks remain the same. Polyspace treats the custom code block in different instances of the referenced model as a single custom code instance.

After extracting the code and model context, Polyspace analyzes them as handwritten code. See “Bug Finder Analysis Assumptions”.

Review Analysis Results

In the Simulink Editor, click **Analysis Results**. The Polyspace User Interface opens with the analysis results. The flagged issues appear in the **Results List** pane. See also:

- “Interpret Bug Finder Results in Polyspace Desktop User Interface” on page 21-2
- “Bug Finder Defect Groups” on page 18-43
- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 22-2
- “Filter and Group Results in Polyspace Desktop User Interface” on page 23-2

To fix the flagged issues, modify the code. For more information, see “Fix Identified Issues” on page 6-49. Alternatively, modify the Simulink model to resolve the Polyspace results. See “Fix Issues” on page 6-38.

If a flagged issue is known or justified, then annotate that information in the custom code blocks. You can annotate the custom code blocks directly from the Polyspace User Interface. See “Annotate Blocks to Justify Results” on page 6-23.

See Also

pslinkoptions | pslinkrun

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 6-2
- “Run Polyspace Analysis on S-Function Code” on page 6-35
- “Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts” on page 6-37
- “Run Polyspace Analysis on Custom Code in C Function Block” on page 6-45
- “Complete List of Polyspace Bug Finder Results”

Run Polyspace Analysis on S-Function Code

If you want to check your S-function code for bugs or errors, you can run Polyspace directly from your S-function block in Simulink.

Prerequisites

Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

S-Function Analysis Workflow

To verify an S-function with Polyspace, follow this recommended workflow:

- 1 Compile your S-function to be compatible with Polyspace.
- 2 Select your Polyspace options.
- 3 Run a Polyspace Bug Finder analysis or a Polyspace Code Prover verification using one of the two analysis modes:
 - This Occurrence — Analyzes the specified occurrence of the S-function with the input for that block.
 - All Occurrences — Analyzes the S-function with input values from every occurrence of the S-function.
- 4 Review results in the Polyspace interface.
 - For information about navigating through your results, see “Filter and Group Results in Polyspace Desktop User Interface” on page 23-2.
 - For help reviewing and understanding the results, see “Complete List of Polyspace Bug Finder Results”.

Compile S-Functions to Be Compatible with Polyspace

Before you analyze your S-function with Polyspace Bug Finder, you must compile your S-function with one of following tools:

- The Legacy Code Tool with the `def.Options.supportCoverageAndDesignVerifier` set to `true`. See `legacy_code`.
- The S-Function Builder block, with **Enable support for Design Verifier** selected on the **Build Info** tab of the S-Function Builder dialog box.
- The Simulink Coverage™ function `slcovmex`, with the option `-sldv`.

Example S-Function Analysis

This example shows the workflow for analyzing S-functions with Polyspace. You use the model `psdemo_model_link_sl` and the S-function `Command_Strategy`.

- 1 Open the model and use the Legacy Code Tool to compile the S-function `Command_Strategy`.

```
% Open Model
psdemo_model_link_sl
```

```

% Compile S-function Command_Strategy
def = legacy_code('initialize');
def.SourceFiles = { 'command_strategy_file.c' };
def.HeaderFiles = { 'command_strategy_file.h' };
def.SFunctionName = 'Command_Strategy';
def.OutputFcnSpec = 'int16 y1 = command_strategy(uint16 u1, uint16 u2)';
def.IncPaths = { fullfile(polyspaceroot, ...
    'toolbox','polyspace','pslink','pslinkdemos','psdemo_model_link_sl') };
def.SrcPaths = def.IncPaths;
def.Options.supportCoverageAndDesignVerifier = true;
legacy_code('compile',def);

```

- 2 Open the model `psdemo_model_link_sl/controller`.
- 3 Specify the code analysis options. On the **Apps** tab, select **Polyspace Code Verifier**. Then, on the **Polyspace** tab:
 - Select the product to run: **Bug Finder** or **Code Prover**. A Code Prover analysis detects run-time errors while a Bug Finder analysis detects coding defects and coding rule violations.
 - Select **Settings**. In the Configuration Parameters dialog box, make sure that the following parameters are set:
 - **Settings from** — Select `Project` configuration. Other values in the drop down menu enables checking different coding rules, which require using **Bug Finder** as the **Mode**.
 - **Open results automatically after verification** — On

Apply your settings and close the Configuration Parameters.
- 4 Right-click the `Command_Strategy` block and select **Polyspace > Verify S-Function > This Occurrence**.
- 5 Follow the analysis in the MATLAB Command Window. When the analysis is finished, your results open in the Polyspace interface.

See Also

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 6-2
- “Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts” on page 6-37

Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts

You can check for bugs and run-time errors in the custom C/C++ code used in your Simulink model. The Polyspace analysis checks functions called from C Caller blocks and Stateflow charts with inputs from the model.

Prerequisites

Before you run Polyspace from Simulink, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

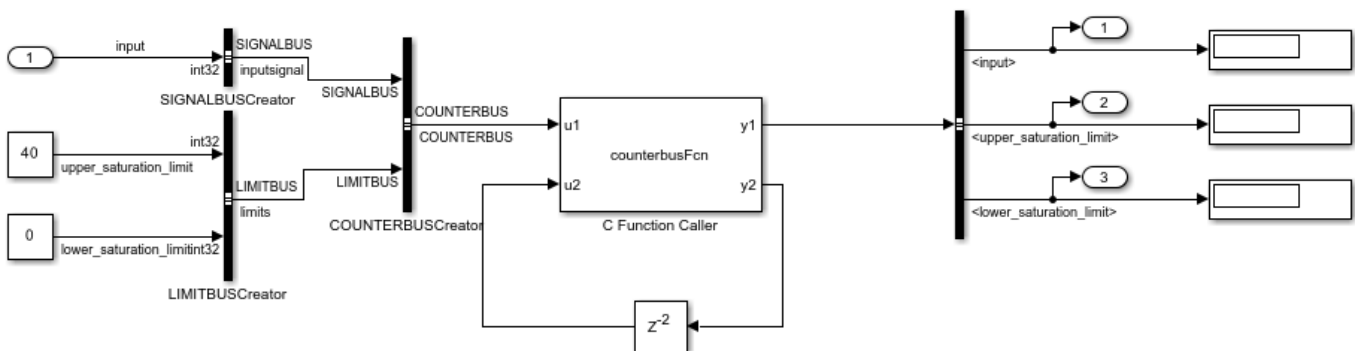
To open the models used in this example, look for this example in the MATLAB Help browser and click the **Open Model** buttons.

C/C++ Function Called Once in Model

This example uses a function called only once in the model from a C Caller block. The analysis checks the function using inputs to the C Caller block.

Open Model for Running Analysis on Custom Code

Open the model `mSlccBusDemo` for analyzing custom code with Polyspace. The model contains a C Caller block that calls a function `counterbusFcn` defined in a file `hCounterBus.c` (declared in file `hCounterBus.h`). The model uses variables saved in a MAT file `dLctData.mat`, which is loaded in the model using a callback.



Copyright 2018 The MathWorks, Inc.

Run Analysis

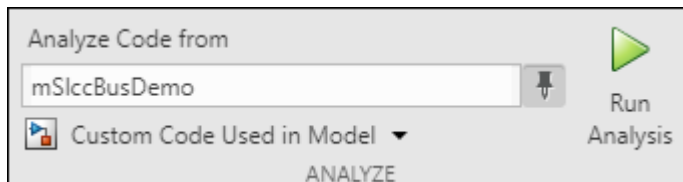
Configure analysis options and run Polyspace.

- 1 On the **Apps** tab, select **Polyspace Code Verifier** to open the **Polyspace** tab.
- 2 Specify the type of analysis:

- Select the product to run, **Bug Finder** or **Code Prover**. A Code Prover analysis detects runtime errors while a Bug Finder analysis detects coding defects and coding rule violations.
- Specify that the analysis must run on custom code in the model instead of generated code.

The **Analyze Code from** field shows the model name. Below the field, instead of **Code Generated as Top Model**, select **Custom Code Used in Model**.

3 Select **Run Analysis**.



Follow the progress of analysis in the MATLAB Command Window. After the analysis, on the **Polyspace** tab, select **Analysis Results**. The results open in the Polyspace user interface.

You can also run the same analysis from MATLAB as follows. The script includes commands to load the model and the `.mat` file containing variables used in the model.

```
openExample('polyspace_code_prover/OpenModelForRunningAnalysisOnCustomCodeExample');
load_system('mSlccBusDemo');
load('dLctData.mat');
```

```
mlopts = pslinkoptions('mSlccBusDemo');
mlopts.VerificationMode = 'CodeProver';
pslinkrun('-slcc','mSlccBusDemo',mlopts);
```

Fix Issues

The analysis results appear on the **Results List** pane in the Polyspace user interface. Select each result and see further details on the **Result Details** pane and the corresponding source code on the **Source** pane.

The rest of this tutorial shows how to investigate and fix issues found in a Code Prover analysis. Similar steps can be followed for issues found with Bug Finder.

If you run a Code Prover analysis, the results contain an orange **Overflow** check.

Family	File	Function	Status
-Run-time Check		1 37	
-Orange Check		1	
-Overflow		1	
?	hCounterBus.c	counterbusFcn()	Unreviewed
-Green Check		37	

The check highlights an addition operation in the `counterbusFcn` function that can overflow:

```
limit = u1->inputsignal.input + u2;
```

The operands come from inputs to `counterbusFcn`, which in turn come from these inputs to the C Caller block:

- The bus `COUNTERBUS`, which combines the signals `input`, `upper_saturation_limit`, and `lower_saturation_limit`. The signal `input` is unbounded.
- The feedback from the C Caller block itself through a Delay block.

You can constrain the signal named `input` in several ways. For instance, you can constrain the `Simulink.Bus` object named `SIGNALBUS` that contains `input`:

- 1 In the Simulink Toolstrip, on the **Modeling** tab, in the **Design** gallery, click **Type Editor**.

The base workspace contains a `Simulink.Bus` object named `SIGNALBUS`.



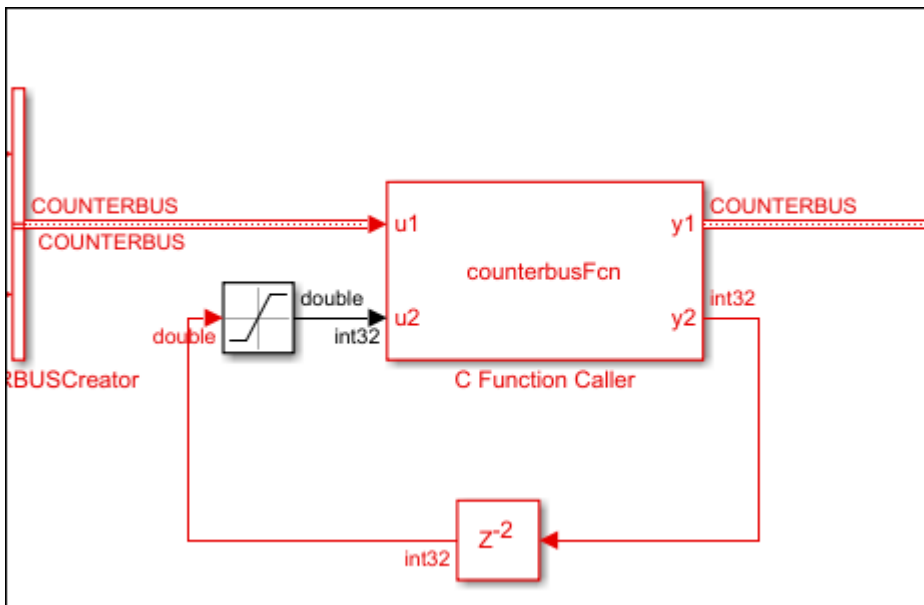
The screenshot shows the 'Contents of Base Workspace' window with a table listing the elements of the `SIGNALBUS` object.

Name	Type	Complexity	Dimensions	DimensionsMode	Min	Max	Unit
> ≡ COUNTERBUS							
> ≡ LIMITBUS							
∨ ≡ SIGNALBUS							
— input	int32	real	1	Fixed			

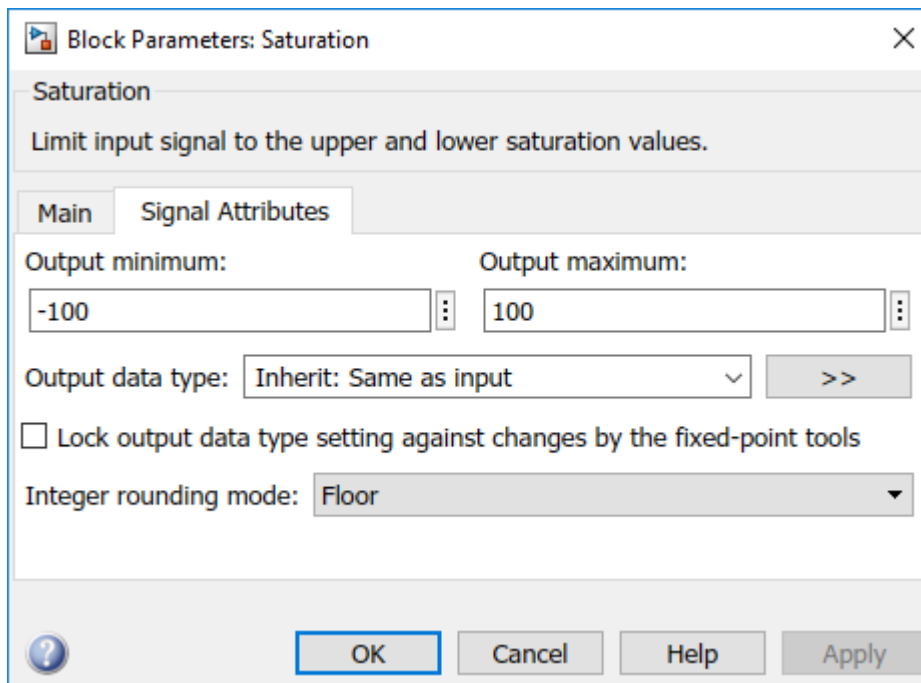
- 2 Specify a minimum and maximum value for the `input` element of `SIGNALBUS`.
- 3 Save the bus object in a MAT file. You can overwrite the file `dLctData.mat` or create a file.

You can also constrain the feedback from the C Caller block in several ways. For instance, you can saturate the feedback signal:

- 1 Add a Saturation block immediately before the feedback signal is input to the C Caller block.



- 2 On the **Signal Attributes** tab, specify a minimum and maximum value for the Saturation block output.



Note that specifying a lower and upper limit on the **Main** tab of the Saturation block is not sufficient to constrain the signal for the Polyspace analysis. The analysis uses the design ranges specified on the **Signal Attributes** tab.

Rerun the analysis. The **Overflow** check in the new set of results is green.

C/C++ Function Called Multiple Times in Model

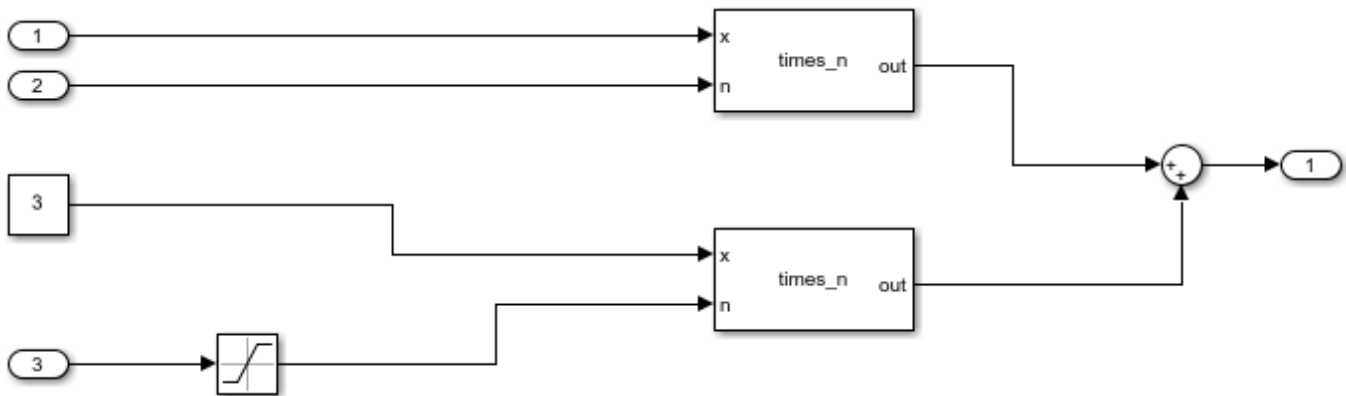
This example uses a function called from multiple C Caller blocks in the model. The function simply returns the product of its two arguments.

The example runs a Code Prover analysis and shows how to determine the function call context starting from Code Prover results. Typically, in a Bug Finder analysis, you do not need to distinguish between different call contexts.

Open Model for Analyzing All Custom Code

Open the model `multiCCallerBlocks` for running Polyspace analysis.

```
openExample('polyspace_bf/OpenModelForAnalyzingAllCustomCodeExample');
open_system('multiCCallerBlocks');
```



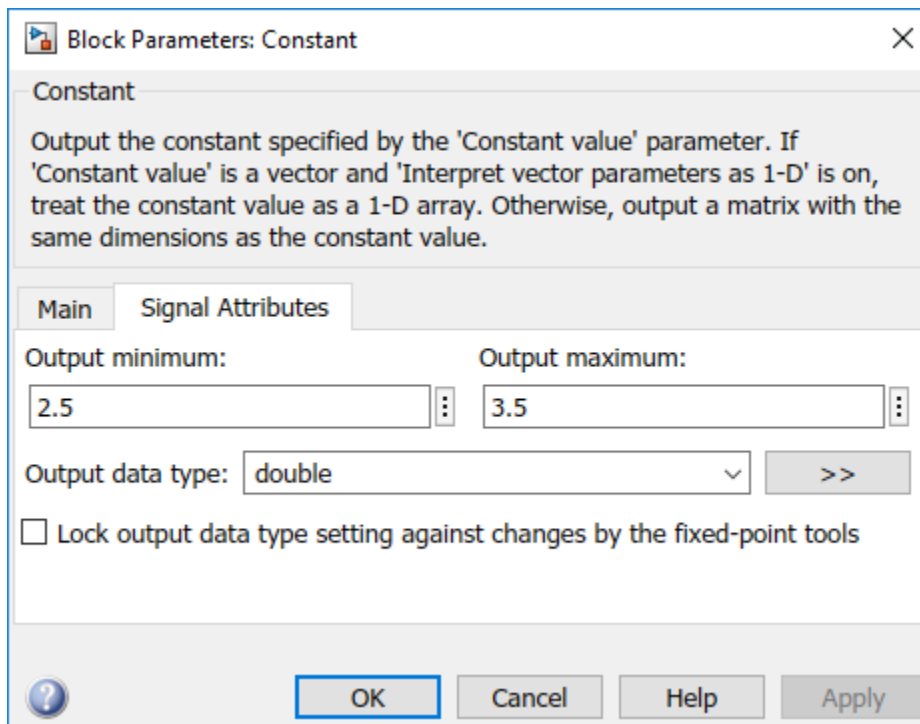
Inspect Model

The model contains two C Caller blocks calling the same function `times_n`. The inputs to one C Caller block come from two Inport blocks that have unbounded input. The inputs to the other C Caller block come from a Constant block and an Inport block that has the input bounded by a Saturation block.

To see the design ranges for the C Caller block that has bounded inputs:

- Double-click the Constant block or the Saturation block.
- On the **Signal Attributes** tab, note the design range.

For instance, although the Constant block has the constant value set to 3, the design range for verification is 2.5 to 3.5.



The design range for the **Saturation** block is [-1,1].

Run Analysis and Review Results

Run analysis as in the previous example and open the results.

The **Results List** pane shows an orange **Overflow** check. The product in the `times_n` function overflows.

```
#include "file.h"

double times_n(double x, double n) {
    return x * n;
}
```

Because the `times_n` function is called from two contexts, the orange color combines both contexts and might indicate two possible situations:

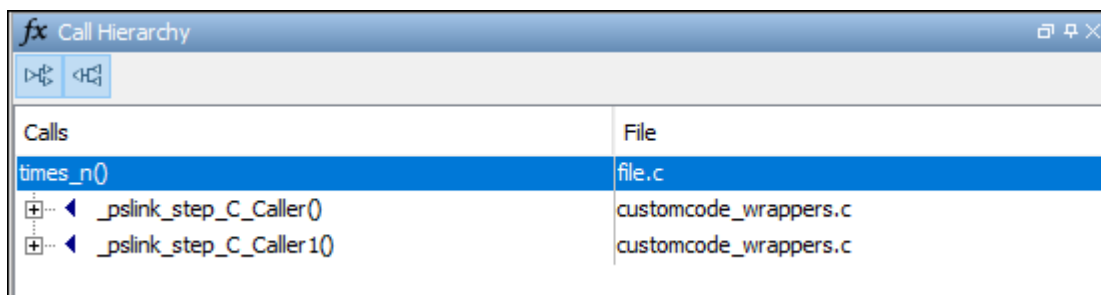
- The overflow occurs in both call contexts.
- The overflow is proven to not occur in one context (green check) and might occur in the other context (orange check).

To determine which call context leads to the overflow:

- 1 See all the callers of `times_n`.

Select the orange **Overflow** check. On the **Result Details** pane, click . The **Call Hierarchy** pane shows the callers of `times_n`.

- 2 On the **Call Hierarchy** pane, you see two wrapper functions as callers. Each wrapper function represents a C Caller block in the model.



Calls	File
times_n()	file.c
└─> _pslink_step_C_Caller()	customcode_wrappers.c
└─> _pslink_step_C_Caller1()	customcode_wrappers.c

Select one of the wrapper functions to open the source code for `customcode_wrappers.c`.

- 3 On the **Source** pane, inspect the code for the wrapper functions. To determine which inputs lead to the overflow, use tooltips on underlined inputs.

For instance, the wrapper function for the C Caller block that has bounded inputs looks similar to this code:

```
/* Go to model '<Root>/C Caller1' */
/* Variables corresponding to inputs for block C Caller1 */
real64_T _pslink_C_Caller1_In1;
real64_T _pslink_C_Caller1_In2;
/* Variables corresponding to outputs for block C Caller1 */
```

```

real64_T _pslink_C_Caller1_Out1;
/* Wrapper functions for code in block C Caller1 */
void _pslink_step_C_Caller1(void) {
    /* See tooltips on function inputs for input ranges */
    _pslink_C_Caller1_Out1 = times_n(_pslink_C_Caller1_In1, _pslink_C_Caller1_In2);
}

```

Use tooltips on the variables to determine their ranges. For instance, the tooltip on the variable `_pslink_C_Caller1_In1` shows that it is in the range [2.5, 3.5] and the tooltip on `_pslink_C_Caller1_In2` shows that it is in the range [-1,1]. Therefore, the product of the two inputs cannot overflow. The overflow must come from the other call context. You can see the tooltips on the inputs to the other call and confirm that the variables are unbounded.

To locate the C Caller block corresponding to a wrapper function, on the **Source** pane, click the blue block name link above the wrapper function (on the line starting with `Go to model`). The C Caller block is highlighted in the model.

Enable Context Sensitivity and Rerun Analysis


In this example, the function is simple enough that you can determine which call context leads to the overflow from the function inputs themselves. For more complex functions, you can configure the analysis to show results from the two contexts separately.

Because distinguishing call contexts involves a deeper analysis, the analysis might take longer. Therefore, enable context sensitivity only for specific functions and only if you are not able to distinguish the call contexts by inspection.

In this example, to enable context sensitivity for the `times_n` function:

- 1 In your model, on the **Polyspace** tab, select **Settings > Project Settings**.

Alternatively, in the Polyspace user interface, select the **Project Browser**. Open the configuration of the project created for the analysis.

- 2 On the **Code Prover Verification > Precision** node, select `custom` for the option **Sensitivity context**. In the **Procedure** field, click  and enter `times_n`.

See also `Sensitivity context (-context-sensitivity)`.

Rerun the analysis from the model and reopen the results. Select the orange **Overflow** check.

The **Result Details** pane shows the call contexts separately. You can see that the overflow occurs only for the call with unbounded inputs (row with orange text) and does not occur for the other call (row with green text).

Click the row with orange text to directly navigate to the wrapper function leading to the orange check. From the wrapper function, you can navigate to the C Caller block with unbounded inputs.

? Overflow ?			
Warning: operation [*] on float may overflow (on MIN or MAX bounds of FLOAT64)			
Calling context	File	Scope	Line
operator * on type float 64 left: full-range [-1.7977E+308 .. 1.7977E+308] right: full-range [-1.7977E+308 .. 1.7977E+308]	customcode_wrappers.c	_pslink_step_C_Caller	26
operator * on type float 64 left: [2.5 .. 3.5] right: [-1.0 .. 1.0] result: [-3.5 .. 3.5]	customcode_wrappers.c	_pslink_step_C_Caller1	38

See Also

`pslinkoptions` | `pslinkrun`

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 6-2
- “Run Polyspace Analysis on S-Function Code” on page 6-35

Run Polyspace Analysis on Custom Code in C Function Block

You can run a Polyspace analysis on the custom C code in a C Function block from Simulink. Polyspace checks the custom C code for errors and bugs while keeping the model specific information such as design range specification, nature and number of inputs that are specified in the Simulink model.

Prerequisites

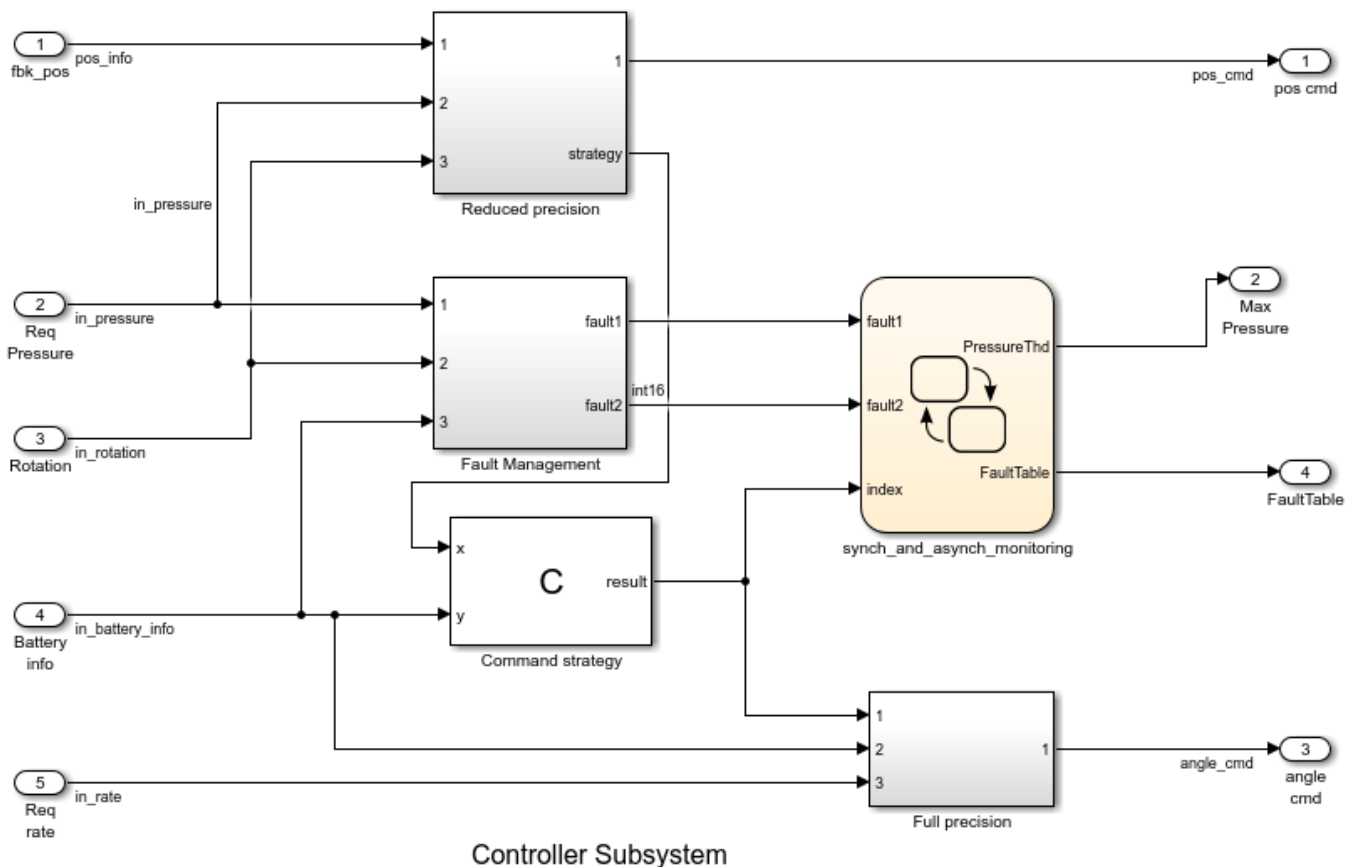
Before you run Polyspace from Simulink, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

To open the model used in this example, in the MATLAB Command Window, run:

```
openExample('polyspace_code_prover/CScriptDemoExample')
open_system('psdemo_model_link_sl_cscript');
```

Open Model for Running Polyspace Analysis on Custom Code in C Function Block

The model contains a C Function block called Command Strategy inside the controller subsystem.



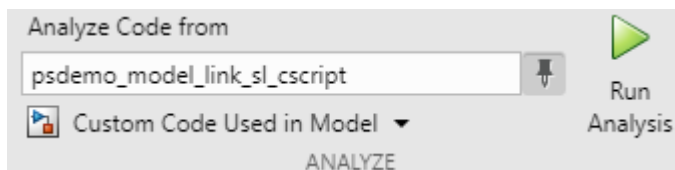
The `Command Strategy` block implements a look-up table using custom C code and outputs a value result based on two inputs `x` and `y`.


Run Polyspace Analysis

Run Polyspace Analysis from Simulink Editor

Click the **Apps** tab and select **Polyspace Code Verifier** to open the **Polyspace** tab.

- 1 Select **Bug Finder** or **Code Prover** in the **Mode** section of the **Polyspace** tab. A **Code Prover** analysis detects run-time errors while a **Bug Finder** analysis detects coding defects and coding rule violations.
- 2 To run a Polyspace analysis on the custom C code in the C Function block, select **Custom Code Used in Model** from the drop-down list in the **Analyze** section.



- 3 To start the Polyspace analysis, click the **Run Analysis** button. The MATLAB Command Window displays the progress of the analysis.
- 4 After the analysis, the Polyspace user interface opens with the results. You can choose to not open the results automatically after the analysis by unselecting **Open results automatically after verification** in **Settings**. To open the results after the analysis is finished, click the **Analysis Results** button.
- 5 To see all results of the Polyspace analysis, click **Clear active filters** from the **Showing** drop-down list in the **Results List** pane. If you run a **Code Prover** analysis, the results for the controller subsystem contain two red checks and an orange check.
- 6 To organize the results by family, click  and select **Family**.

Family	Information	File
[-] Run-time Check		2 1 23
[-] Red Check		2
[-] Illegally dereferenced pointer		1
[-] Out of bounds array index		1
[-] Orange Check		1
[-] Overflow		1
[-] Green Check		23
[-] Global Variable		
[-] Not shared		

To switch between a **Bug Finder** and **Code Prover** analysis, return to the Simulink Editor from the Polyspace user interface. Switch between **Bug Finder** and **Code Prover** in the **Mode** section and run the analysis again.

Run Polyspace Analysis from MATLAB

You can run a Polyspace Code Prover analysis on the custom code for this model from MATLAB Editor or the Command Window using this code:

```
% Load the model 'psdemo_model_link_sl_cscript'
load_system('psdemo_model_link_sl_cscript');
% Create a 'pslinkoptions' object
mlopts = pslinkoptions('psdemo_model_link_sl_cscript');
% Specify whether to run 'CodeProver' or 'BugFinder' Analysis
mlopts.VerificationMode = 'CodeProver';
% Specify custom code as analysis target and run the analysis
pslinkrun('-slcc', 'psdemo_model_link_sl_cscript', mlopts);
```

Identify Issues in C Code

To identify issues in the custom C code, use the information in the **Result Details** pane and the **Source** pane of the Polyspace user interface. If you do not see these panes, go to **Window > Show/Hide View** and select the missing pane. For details on the panes, see “Result Details in Polyspace Desktop User Interface” on page 21-22 and “Source Code in Polyspace Desktop User Interface” on page 21-17.

Identify C Function Block Inputs and Outputs in Source Pane

Polyspace wraps the code in the C Function block in a custom code wrapper. The inputs and outputs of the C Function block are declared as global variables. The custom C code is called as a function.

```
/* Variables corresponding to inputs ..*/
// global In...
/* Variables corresponding to outputs*/
// global Out...
/* Wrapper functions for code in block */
// void ...(void){
//     //...
// }
}
```

- The global variables corresponding to inputs start with **In**, such as `In1_psdemo_model_link_sl_cscript_98_Command_strategy`.
- The global variables corresponding to outputs start with **Out**, such as `Out1_psdemo_model_link_sl_cscript_98_Command_strategy`.
- The `void-void` function contains the custom C code with the input and output variables replaced by the global variables. If you have multiple C Function blocks, then the code in each block is wrapped in separate functions.

The global variables reflect all properties of the input and output of the C Function block, including their data range, data type, and size. If you have multiple inputs, then the order of the global variables is the same as the order of the input defined in the C Function block. This table shows the input and output variables of the block in this example and their corresponding global variables in the **Source** pane.

Global Variable Name in Source Pane	Scope	Variable Name in C Function Block
In1_psdemo_model_link_sl_cscript_98_Command_strategy	Input	x
In2_psdemo_model_link_sl_cscript_98_Command_strategy	Input	y
Out1_psdemo_model_link_sl_cscript_98_Command_strategy	Output	result

Identify issues in the custom code by reviewing the wrapped code in the **Source** pane. Use the tooltip in the **Source** pane and the information in the **Result Details** pane to fix the issues. This workflow applies to **Code Prover** and **Bug Finder** analyses.

Illegally dereferenced pointer

The red check **Illegally dereferenced pointer** highlights the dereferencing operation after the for loop.

```
tmp = *p + 5;
```

The **Result Details** pane states that the pointer `*p` is outside its bounds. To find the root cause of the check, follow the life cycle of the pointer leading to the illegal dereferencing.

- 1 At the start of its life cycle, the pointer `*p` points to the first element of array which has 100 elements.
- 2 Then `p` is incremented 100 times, pointing `*p` to the nonexistent location `array[100]`.
- 3 The dereferencing operation in `tmp = *p+5;` becomes illegal, causing a red check.

Out of Bounds array index

The red check **Out of Bounds array index** highlights the array indexing operation in the `if` condition.

```
if (another_array[return_val - i + 9] != 0)
```

The **Result Details** pane states that the size of `another_array` is 2 while the index value `return_val - i + 9` ranges from 2 to 18. To find the root cause of the check, track the values of the variables `return_val` and `i` using the tooltip. When you hover over any instance of the variables in the **Source** pane, the tooltip is displayed.

- 1 The value of `i` is 100.
- 2 The value of `return_val` ranges from 93 to 109 because of the prevailing condition: `if ((return_val > 92) && (return_val < 110))`.
- 3 The index value `(return_val - i + 9)` evaluates to a range of 2 to 18.
- 4 The index values are out of bounds for the array `another_array`, causing a red check.

Overflow

The orange **Overflow** check highlights the assignment to `return_val`. The **Result Details** pane states that the check is related to bounded input values. To find the root cause of the check, check the data type and corresponding range of the variables by using the tooltip.

- The input values `x` and `y` correspond to these respective global variables
 - `In1_psdemo_model_link_sl_cscript_98_Command_strategy`
 - `In2_psdemo_model_link_sl_cscript_98_Command_strategy`
- The first input `x` is an unbound unsigned integer. Because `x` is unbound, it has the full range of an unsigned integer, which is from 0 to 65535.
- The second input `y` is a bounded unsigned integer ranging from 0 to 1023.
- `x-y` is assigned to the unbound signed integer `return_val`. Because `return_val` is unbound, it has full range from -32768 to 32767.
- The range of `x-y` is 1023 to 65535, while the range of `return_val` is -32768 to 32767.
- Some possible values of `x-y` cannot fit into `return_val`, causing the orange check.

For details about interpreting results of a Polyspace Bug Finder analysis, see “Interpret Bug Finder Results in Polyspace Desktop User Interface” on page 21-2.

Fix Identified Issues

Modify the custom C code or the model to fix the issues. You can fix a Polyspace check in several ways. The examples here illustrate the general workflow of fixing Polyspace checks.

Illegally dereferenced pointer

You can address this check in several ways. Modify the C code so that a nonexistent memory address is not accessed.

- 1 Return to the Simulink Editor and double-click on the C Function block to open the custom code.
- 2 Use the index operator on `array` to access a valid array index. You can access indices from 0 to 99 because `array` has 100 elements. Accessing indices beyond this range results in a run-time error in Simulink.

```
// access any index between 0 to 99
tmp = array[50] + 5;
```

Alternatively, assign the address of a valid memory location to `p` before the dereferencing operation. For example, `*p` can point to the 51st element in `array`.

```
// After the for loop, point p to a valid memory location
p = &(array[50]);
// ...
tmp = *p + 5;
```

Out of Bounds array index

You can address this check in several ways. Modify the code so that the size of `another_array[]` remains larger than or equal to the index value `return_val - i + 9`.

- 1 Return to the Simulink Editor and double-click on the C Function block to open the custom code.
- 2 Modify the prevailing condition on `return_val` so that the index value `return_val - i + 9` always evaluates to 0 or 1.

```
if ((return_val > 91) && (return_val < 93))
//...
```

Alternatively, declare `another_array` with size 19.

```
int another_array[19];
```

Overflow

You can address this check in several ways as well. Modify the C code or the model so that the range of the right side of the assignment operation remains equal to or larger than that of the left side.

- 1 Return to the Simulink Editor.
- 2 Saturate the input variables `x` and `y` in the model so that their difference can fit into a 16-bit integer. The workflow for fixing **Overflow** by using saturation blocks is described in “Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts” on page 6-37.

Alternatively, increase the size of `return_val` in the custom C code to accommodate `x - y`.

- 1 Return to the Simulink Editor and double-click on the C Function block to open the custom code.
- 2 Declare `return_val` as a 32-bit integer.

```
int32_T return_val;
```

For details about addressing Polyspace results, see “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 22-2.

See Also

`pslinkoptions` | `pslinkrun`

More About

- “Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts” on page 6-37
- “Complete List of Polyspace Bug Finder Results”

Recommended Model Configuration Parameters for Polyspace Analysis

For Polyspace analyses, set the following configuration parameters before generating code. If you do not use the recommended value for `SystemTargetFile`, you get an error. For other parameters, if you do not use the recommended value, you get a warning.

Grouping	Command-Line	Name and Location in Configuration
Code Generation	Name: <code>SystemTargetFile</code> (Simulink Coder) Value: An Embedded Coder Target Language Compiler (TLC) file. For example <code>ert.tlc</code> or <code>autosar.tlc</code> .	Location: Code Generation Name: System target file Value: Embedded Coder target file
	Name: <code>MatFileLogging</code> (Simulink Coder) Value: 'off'	Location: Code Generation > Interface Name: MAT-file logging Value: <input type="checkbox"/> Not selected
	Name: <code>GenerateReport</code> (Simulink Coder) Value: 'on'	Location: Code Generation > Report Name: Create code-generation report Value: <input checked="" type="checkbox"/> Selected
	Name: <code>IncludeHyperlinkInReport</code> (Embedded Coder) Value: 'on'	Location: Code Generation > Report Name: Code-to-model Value: <input checked="" type="checkbox"/> Selected
	Name: <code>GenerateSampleERTMain</code> (Embedded Coder) Value: 'off'	Location: Code Generation > Templates Name: Generate an example main program Value: <input type="checkbox"/> Not selected
	Name: <code>GenerateComments</code> (Simulink Coder) Value: 'on'	Location: Code Generation > Comments Name: Include comments Value: <input checked="" type="checkbox"/> Selected

Grouping	Command-Line	Name and Location in Configuration
Optimization	Name: DefaultParameterBehavior (Simulink Coder) Value: 'Inlined'	Location: Optimization Name: Default parameter behavior Value: Inlined
	Name: InitFltsAndDblsToZero (Simulink Coder) Value: 'on'	Location: Optimization Name: Use memset to initialize floats and doubles to 0.0 Value: <input type="checkbox"/> Not selected
	Name: ZeroExternalMemoryAtStartup (Embedded Coder) Value: 'off'	Location: Optimization Name: Remove root level I/O zero initialization Value: <input checked="" type="checkbox"/> Selected
Solver	Name: SolverType (Simulink) Value: 'Fixed-Step'	Location: Solver Name: Type Value: Fixed-step
	Name: Solver (Simulink) Value: 'FixedStepDiscrete'	Location: Solver Name: Solver Value: discrete (no continuous states)

Configure Polyspace Options in Simulink

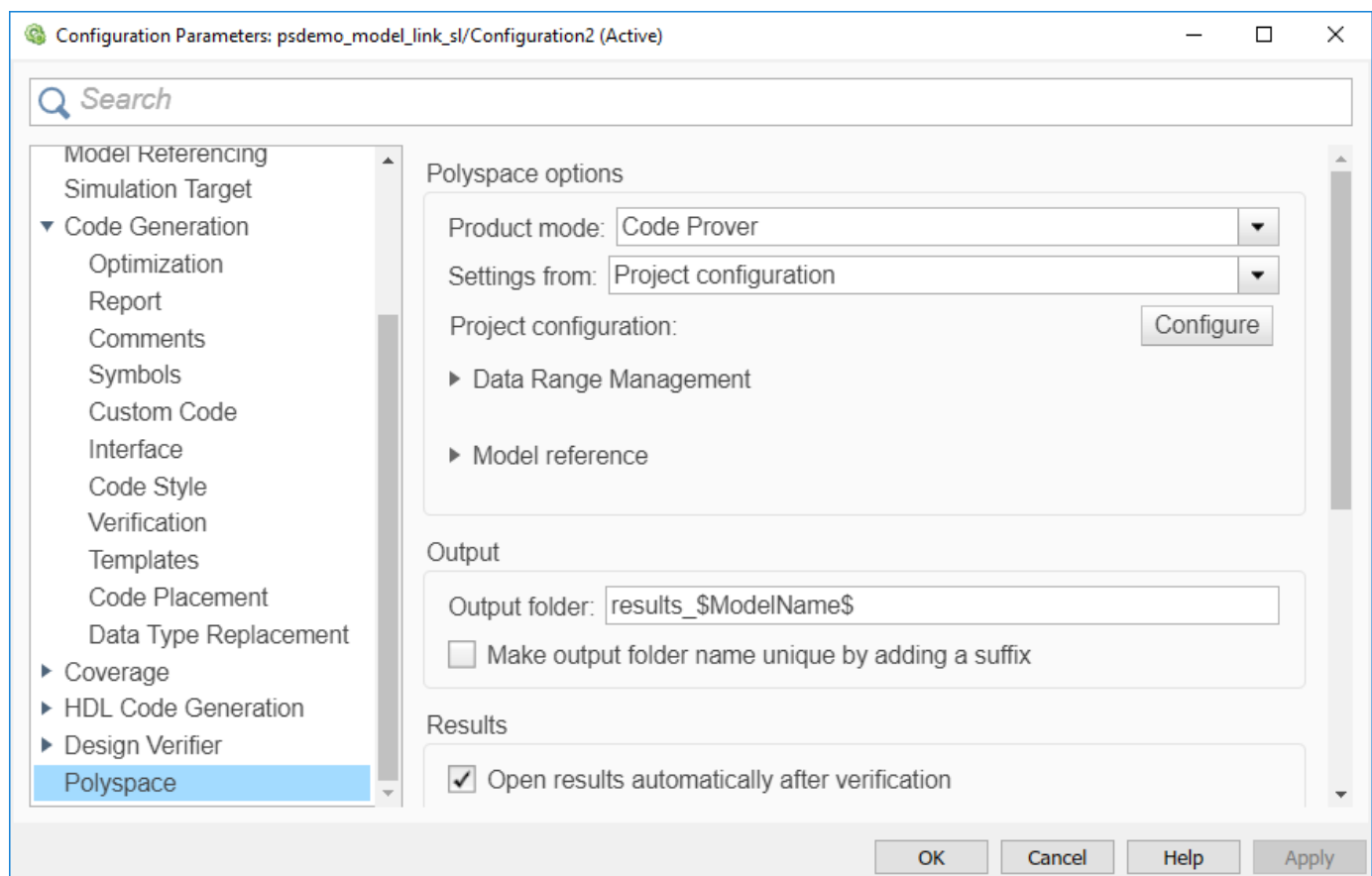
Configure basic and advanced Polyspace options when analyzing generated code. You can reuse existing configuration across multiple analysis.

To get started with Polyspace analysis in Simulink, see “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 6-2.

Configure Options

Set basic options

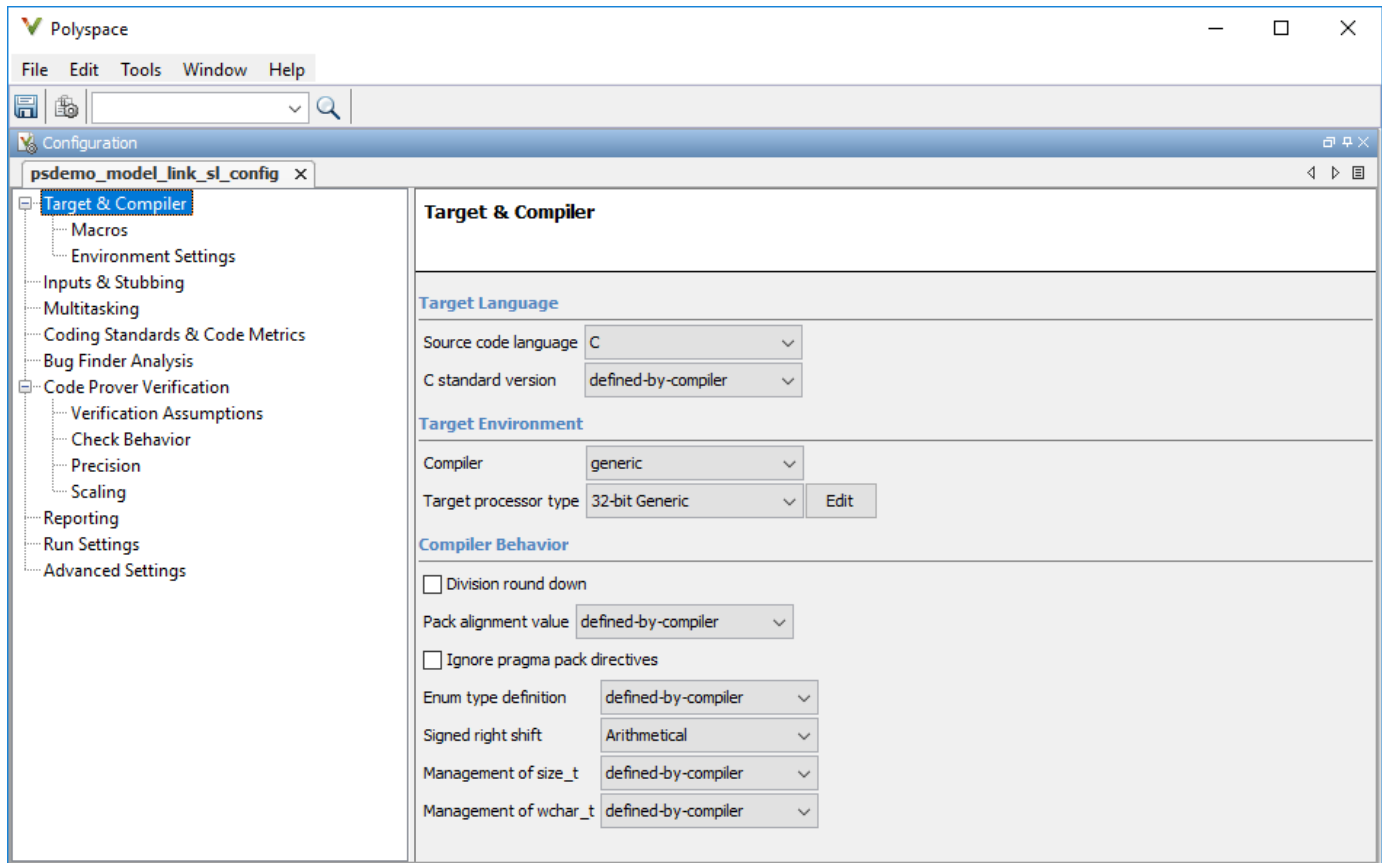
To set the basic Polyspace options in the Simulink Configuration Parameters window, on the **Apps** tab, select **Polyspace Code Verifier**. Then, on the **Polyspace** tab, select **Settings** or **Settings > Polyspace Settings**.



Set advanced options

The advanced options appear on the **Configuration** pane that also appears in the Polyspace user interface when you manually create a project for handwritten code.

To open the advanced options, on the **Polyspace** tab, select **Settings > Project Settings**.



On this pane, you can specify advanced settings.

- In the **Run Settings** pane, select options to run the code analysis on a remote cluster. Alternatively, in the **Advanced Settings** pane, use the option Run Bug Finder or Code Prover analysis on a remote cluster (-batch) in the **Other** field.

If you use this option, after starting the analysis, you can follow the analysis progress on the remote cluster through the Job Monitor window. On the **Polyspace** tab, select **Remote Job Monitor**.

- In the **Inputs & Stubbing** pane, specify options to stub certain functions for the analysis and then constrain the function output. Alternatively, in the **Advanced Settings** pane, use the options Functions to stub (-functions-to-stub) and Constraint setup (-data-range-specifications) in the **Other** field.

If a basic option in the Configuration Parameters window directly conflicts with an advanced option in the Polyspace window, the former prevails. For instance, say you specify these options:

- “Settings from (C)”: You select this basic option Project configuration and MISRA C 2012 checking for generated code.
- Check MISRA C:2012 (-misra3): You disable this advanced option.

Polyspace ignores the advanced option and checks for violations of MISRA C:2012 rules.

By default, the advanced options are saved in the project file *modelname_config.psprj* in the *pslink_config* subfolder of the *results* folder. Use this project file to reuse the options associated with the project..


Share and Reuse Configuration

Share the basic or advanced options across multiple models.

- Basic options — Share and reuse the options set in the Configuration Parameters window. See “Share a Configuration with Multiple Models” (Simulink).
- Advanced options — Share and reuse the advanced options that are in a separate Polyspace project. Share this project across multiple models. When reusing advanced Polyspace options that are saved in a Polyspace project file, use a project file that is configured by using the **Polyspace App** in the Simulink Editor, as shown in “Set advanced options” on page 6-53. Reusing a project file that is not generated from the Simulink Editor can result in unexpected results.

You can specify the advanced options once, and then reuse the advanced options across multiple models. Set the basic options in each model individually.

Set options from model

Set the advanced options as needed. To see where the associated project file is stored or to change the file name, on the Polyspace window toolbar, click the  icon.

Reuse options in another model

To reuse the advanced options in another model, open the open the model and open the Configuration Parameters window. On the **Polyspace** tab, select **Settings**.

- Select **Use custom project file**. Provide the path to the *.psprj project file that you previously created.
- To use the project settings, select Project configuration under **Settings from**.

If you want to check for additional issues, such as MISRA C: 2012 violations, select the options Project configuration and MISRA C 2012 checking for generated code.

If you run an analysis from the command line, you can set these options with the `pslinkoptions` function. See also `pslinkoptions` Properties.

See Also

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 6-2
- “Run Polyspace Analysis on Code Generated with TargetLink” on page 6-62
- “Default Polyspace Options for Code Generated with Embedded Coder” on page 6-57
- “Default Polyspace Options for Code Generated with TargetLink” on page 6-64

How Polyspace Analysis of Generated Code Works

When you generate code from a Simulink model, the generated code can contain these components:

- `initialize()` functions that run before the simulation starts.
- `terminate()` functions that run after the simulation ends.
- `step()` functions that run in a loop to perform the simulation.

Additionally, the generated code might have a placeholder `main()` function that contains calls to the above. You might edit the placeholder `main()` to fit your deployment purposes. For more information about the `main` generated by Embedded Coder, see “Main Program” (Embedded Coder).

When you run Polyspace on generated code, Polyspace gathers this information from your code:

- `initialize()` functions
- `terminate()` functions
- `step()` functions
- List of parameter variables
- List of input variables

When you run Code Prover, the software uses this information to generate a separate `main()` function to facilitate the analysis. Regardless of the presence of the generated placeholder `main()`, Polyspace uses its own `main()` function that performs these tasks:

- 1 Initializes parameters by using the Polyspace option `Parameters (-variables-written-before-loop)`.
- 2 Calls initialization functions by using the option `Initialization functions (-functions-called-before-loop)`.
- 3 Initializes inputs using the option `Inputs (-variables-written-in-loop)`.
- 4 Calls the `step` function in a loop by using the option `Step functions (-functions-called-in-loop)`. By default, Polyspace assumes that the `step` function might be called an arbitrary number of times in the loop. To specify the number of iterations in the loop for a more precise Code Prover analysis, use the option `-main-generator-bounded-loop`.
- 5 Calls the `terminate` function by using the option `Termination functions (-functions-called-after-loop)`.

The Polyspace generated `main` function might have this structure:

```
init parameters    \\ -variables-written-before-loop
init_fct()        \\ -functions-called-before-loop
  while(random){   \\ start main loop with one or more iterations
    init inputs    \\ -variables-written-in-loop
    step_fct()     \\ -functions-called-in-loop
  }
terminate_fct()   \\ -functions-called-after-loop
```

For C++ code generated with Embedded Coder, the `initialize()`, `step()`, and `terminate()` functions and associated variables are either class members or have global scope.

Default Polyspace Options for Code Generated with Embedded Coder

In this section...

“Default Options” on page 6-57

“Constraint Specification” on page 6-57

“Recommended Polyspace options for Verifying Generated Code” on page 6-58

“Hardware Mapping Between Simulink and Polyspace” on page 6-58

Default Options

For Embedded Coder code, the software sets the following verification options by default:

```
-sources path_to_source_code
-D PST_ERRNO
-D main=main_rtwec
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
-functions-to-stub=[rtIsNaN,rtIsInf,rtIsNaNF,rtIsInfF]
-results-dir results
```

Note *matlabroot* is the MATLAB installation folder.

Constraint Specification

You can constrain inputs, parameters, and outputs to lie within specified ranges. Use these configuration parameters:

- “Input”
- “Tunable parameters”
- “Output”

The software automatically creates a Polyspace constraints file using information from the MATLAB workspace and block parameters.

You can also manually define a constraints file in the Polyspace user interface. See “Specify External Constraints for Polyspace Analysis” on page 14-2. If you define a constraints file, the software appends the automatically generated information to the constraints file you create. Manually defined constraint information overrides automatically generated information for all variables.

The software supports the automatic generation of constraint specifications for the following kinds of generated code:

- Code from standalone models
- Code from configured function prototypes

- Reusable code
- Code generated from referenced models and submodels

Additional Information

See also “External Constraints on Polyspace Analysis of Generated Code” on page 6-59.

Recommended Polyspace options for Verifying Generated Code

For Embedded Coder code, the software automatically specifies values for the following verification options:

- `-main-generator`
- `-functions-called-in-loop`
- `-functions-called-before-loop`
- `-functions-called-after-loop`
- `-variables-written-in-loop`
- `-variables-written-before-loop`

Embedded Coder performs a wraparound of the variables in the generated code that might overflow. When running a Code Prover analysis of code generated by Embedded Coder, Polyspace uses these options:

- `-signed-integer-overflows warn-with-wrap-around`
- `-unsigned-integer-overflows allow`

These options might have different default values when analyzing code that is not generated by Embedded Coder. See **Overflow mode for signed integer** (`-signed-integer-overflows`) and **Overflow mode for unsigned integer** (`-unsigned-integer-overflows`).

In addition, for the option `-server`, the software uses the value specified in the **Send to Polyspace server** check box on the **Polyspace** pane. These values override the corresponding option values in the **Configuration** pane of the Polyspace user interface.

You can specify other verification options for your Polyspace Project through the Polyspace **Configuration** pane. See “Configure Polyspace Options in Simulink” on page 6-53.

Hardware Mapping Between Simulink and Polyspace

The software automatically imports target word lengths and byte ordering (endianness) from Simulink model hardware configuration settings. The software maps **Device vendor** and **Device type** settings on the Simulink **Configuration Parameters > Hardware Implementation** pane to **Target processor type** settings on the Polyspace **Configuration** pane.

The software creates a generic target for the verification.

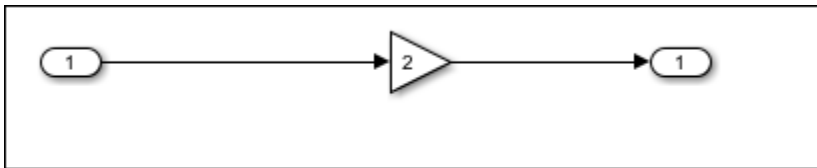
External Constraints on Polyspace Analysis of Generated Code

When you check generated code for bugs or run-time errors, you can choose whether to perform the check for all values of an input or a specific range of values. You can extract the input range from the Simulink model, or specify your own external constraints.

Likewise, you can use a fixed value for tunable parameters or a range of values. You can also check whether output values fall within a specific range.

Extract External Constraints from Model

Consider this simple model with an Inport block, a Gain block, and an Outport block. Suppose the signal in the Inport and Outport blocks and the gain parameter of the Gain block have a minimum and maximum value.



You can analyze the code generated from this model with these minimum and maximum values. On the **Apps** tab, select **Polyspace Code Verifier**. Then, on the **Polyspace** tab, select **Settings**. Specify these configuration parameters:

- “Input”: Select **Use specified minimum and maximum values**. The Code Prover analysis checks the generated code within the specified range of values from the Inport block. The Bug Finder analysis uses this information to exclude false positives.

Default: This option is selected.

- “Tunable parameters”: Select **Use specified minimum and maximum values**.

Default: This option is not selected. The analysis uses the fixed gain value of the Gain block (the value 2 in the example).

For the analysis to consider a range instead of a fixed value, the parameters must be tunable and not inlined. See **Default parameter behavior**.

- “Output”: Select **Verify outputs are within minimum and maximum values**. The Code Prover analysis creates a red check if the outputs exceed the range specified on the Outport block. See also **Correctness condition**.

Default: This option is not selected. The Code Prover analysis does not check output values.

After analysis, to check if a constrained range value is used, see one of these files:

- Constraint specification XML file `modelname_drs.xml` in the folder `results_modelname\modelname`.
- Polyspace project file `modelname.prpsj` in the folder `results_modelname`.

Open this file in the Polyspace user interface. In the project configuration, see the extracted constraints specified for the option **Constraint setup (-data-range-specifications)**.

Storage Classes Supported for Constraint Extraction From Simulink Model

To allow constraint extraction from the Simulink model, the signals and parameters must have data types in specific storage classes. For details on storage classes, see “Choose Storage Class for Controlling Data Representation in Generated Code” (Embedded Coder).

Common Storage Classes

Storage Class	Signal Constraint Supported	Parameter Constraint Supported
Auto	Yes	Yes
ExportedGlobal	Yes	Yes
ImportedExtern	Yes	Yes
ImportedExternPointer	Yes	Yes
Model default	Yes	Yes

Other Storage Classes

Storage Class	Signal Constraint Supported	Parameter Constraint Supported
BitField	Yes	Yes
CompilerFlag	No	No
Const	No	Yes
ConstVolatile	No	Yes
Define	No	No
ExportToFile	Yes	Yes
FileScope	Yes	No
GetSet	No	No
ImportedDefine	No	No
ImportFromFile	No	No
Struct	No	No
Volatile	Yes	Yes

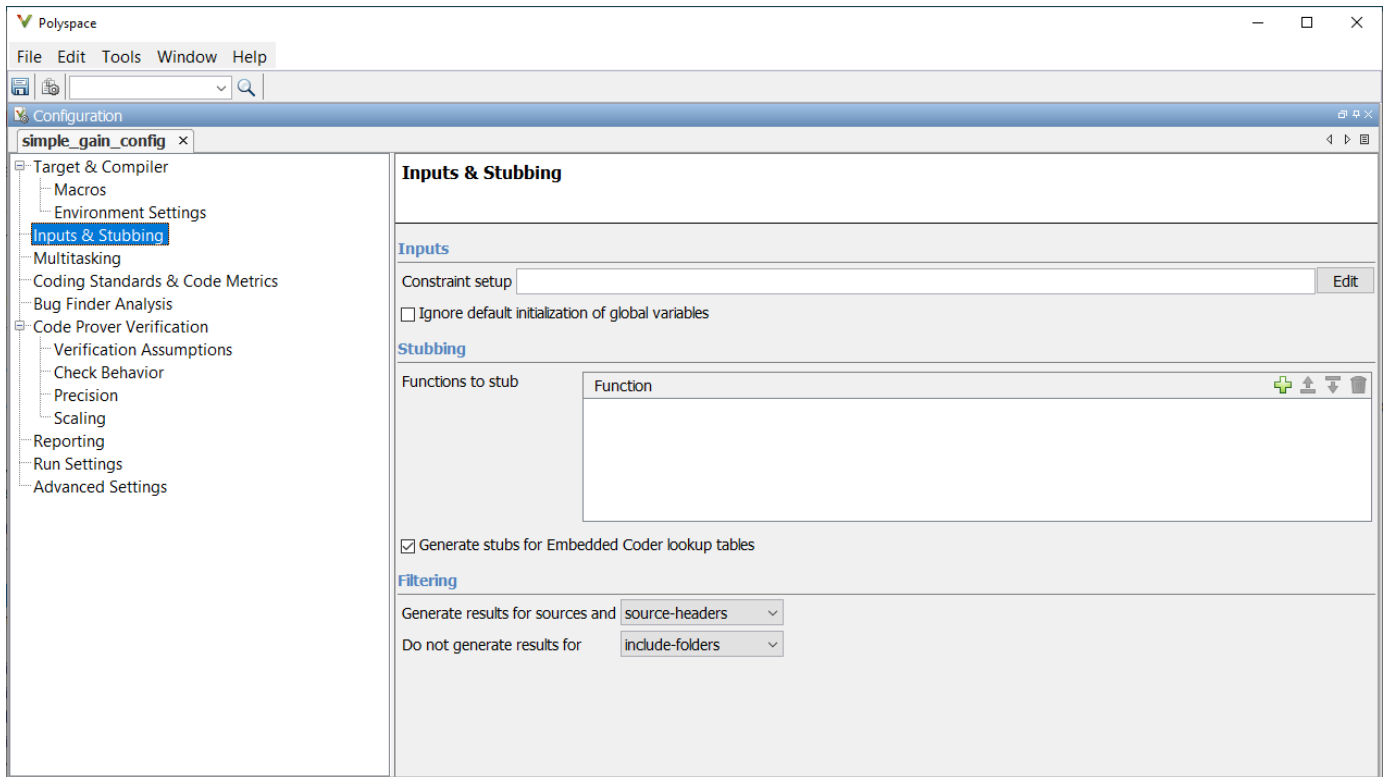
Specify Custom External Constraints

In some instances, you might need to specify a custom set of constraints on your generated code. For instance, you might be integrating the generated code with an existing code base, which imposes a set of custom constraints.

When analyzing the generated code, specify custom external constraints through the Polyspace Configuration window:

- 1 In the Simulink Configuration Parameters window, locate the **Polyspace** tab, and then click **Configure** to open the Polyspace Configuration window.

- 2 In the **Constraint Setup** field, located in the **Inputs & Stubbing** node, specify the custom external specification XML file.



You can create and edit a custom external constraint template through the Polyspace user interface. See “Specify External Constraints for Polyspace Analysis” on page 14-2.

See Also

More About

- “Default Polyspace Options for Code Generated with Embedded Coder” on page 6-57
- “Choose Storage Class for Controlling Data Representation in Generated Code” (Embedded Coder)
- “Specify External Constraints for Polyspace Analysis” on page 14-2
- “External Constraints for Polyspace Analysis” on page 14-6

Run Polyspace Analysis on Code Generated with TargetLink

To detect bugs and runtime errors, run a Polyspace analysis after generating code from Simulink models by using TargetLink. Run the analysis from the Simulink Editor window. Manually setting up a Polyspace project is not necessary. If you use Embedded Coder to generate code, see “Run Polyspace Analysis on Code Generated from Simulink Model” on page 6-15.

Configure and Run Analysis

Configure code analysis

On the **Apps** tab, select **Polyspace Code Verifier**. Then, on the **Polyspace** tab:

- Select the product to run: **Bug Finder** or **Code Prover**. A Code Prover analysis detects run-time errors while a Bug Finder analysis detects coding defects and coding rule violations.
- Select **Settings**. Change default values of these options if needed.
 - “Settings from (C)”: Enable checking of MISRA or JSF® coding rules in addition to the default checks.
 - “Output folder”: Specify a dedicated folder for results. The default analysis runs Code Prover on generated code and saves the results in a folder `results_modelName` in the current working folder.
 - “Enable additional file list”: Add C files that are not part of the generated code.
 - “Open results automatically after verification”

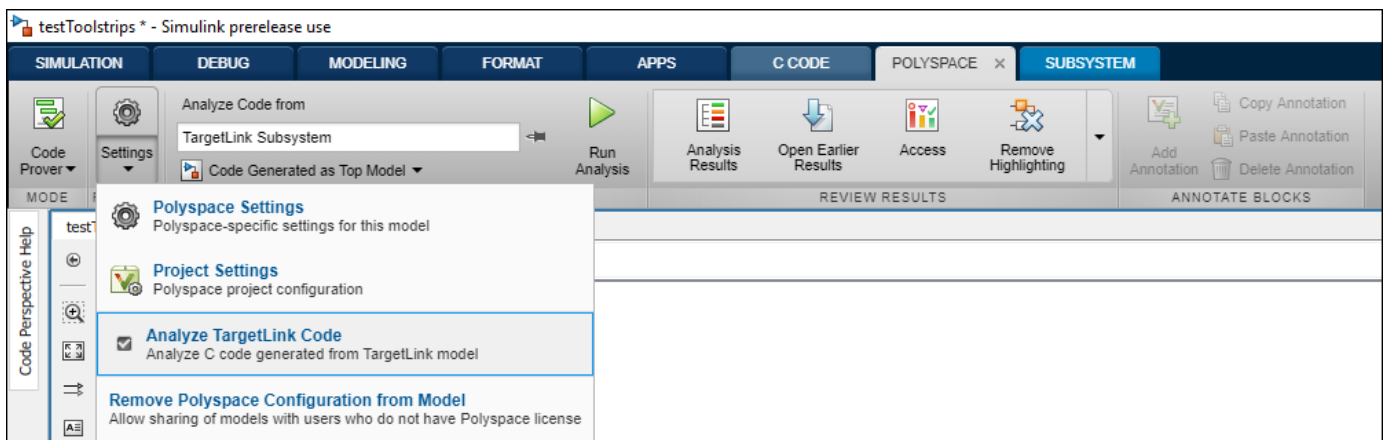
Analyze code

To analyze generated code:

- 1 Choose to analyze code generated from a TargetLink Subsystem. You cannot analyze code generated from the entire model.

The **Analyze Code from** field shows the top model. Unpin the content of this field and then select the TargetLink Subsystem.

- 2 Select **Settings > Analyze TargetLink Code**. Then, select **Run Analysis**.



You can follow the progress of the analysis in the MATLAB command window.

The results open automatically unless explicitly disabled. By default, the results are saved in a folder `results_ModelName` in the current folder. Each new run overwrites previous results. You can change these behaviors or save the results to a Simulink project using appropriate configuration parameters.

Review Analysis Results

Review result in code

The results appear on the **Results List** pane. Click each result to see the source code and details on the **Result Details** pane.

Navigate from code to model

Links in code comments show blocks that generate the subsequent lines of code. To see the blocks in the model, click the block names.

Fix issue

Investigate whether the issues in your code are related to design flaws in the model.

For instance, you might need to constrain the range of signal from Inport blocks. See “Work with Signal Ranges in Blocks” (Simulink). If a flagged issue is known or justified, then annotate that information in the relevant blocks. To annotate a block in Simulink Editor, right-click the block and use the contextual menu.

Default Polyspace Options for Code Generated with TargetLink

In this section...

“TargetLink Support” on page 6-64
 “Default Options” on page 6-64
 “Lookup Tables” on page 6-64
 “Data Range Specification” on page 6-65
 “Code Generation Options” on page 6-65

TargetLink Support

The Windows version of Polyspace Bug Finder is compatible with dSPACE® Data Dictionary and TargetLink Code Generator.

Polyspace Bug Finder does support CTO generated code. However, for better results, MathWorks recommends that you disable the CTO option in TargetLink before generating code. For more information, see the dSPACE documentation.

Because Polyspace Bug Finder extracts information from the dSPACE Data Dictionary, you must regenerate the code before performing an analysis.

Default Options

Polyspace sets the following options by default:

```

-sources path_to_source_code
-results-dir results_folder_name
-I path_to_source_code
-D PST_ERRNO
-I dspaceroot\matlab\TL\SimFiles\Generic
-I dspaceroot\matlab\TL\srcfiles\Generic
-I dspaceroot\matlab\TL\srcfiles\i86\LCC
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
-functions-to-stub=[rtIsNaN,rtIsInf,rtIsNaNF,rtIsInfF]
-scalar-overflows-behavior wrap-around
-boolean-types Bool
  
```

Note *dspaceroot* and *matlabroot* are the dSPACE and MATLAB tool installation directories respectively.

Lookup Tables

By default, Polyspace provides stubs for the lookup table functions. The dSPACE data dictionary is used to define the range of their return values. A lookup table that uses extrapolation returns full range for the type of variable that it returns. You can disable this behavior from the Polyspace configuration menu.

Data Range Specification

You can constrain inputs, parameters, and outputs to lie within specified data ranges. See “Work with Signal Ranges in Blocks” (Simulink).

The software automatically creates a Polyspace constraints file using the dSPACE Data Dictionary for each global variable. The constraint information is used to initialize each global variable to the range of valid values as defined by the min..max information in the data dictionary. This information allows Polyspace software to model real values for the system during analysis. Carefully defining the min-max information in the model allows the analysis to be more precise, because only the range of real values is analyzed.

Note Boolean types are modeled having a minimum value of 0 and a maximum of 1.

You can also manually define a constraint file in the Polyspace user interface. See “Specify External Constraints for Polyspace Analysis” on page 14-2. If you define a constraint file, the software appends the automatically generated information to the constraint file you create. Manually defined constraint information overrides automatically generated information for all variables.

Constraints cannot be applied to static variables. Therefore, the compilation flags `-D static=` is set automatically. It has the effect of removing the static keyword from the code. If you have a problem with name clashes in the global name space, either rename the variables or disable this option in Polyspace configuration.

Code Generation Options

From the TargetLink Main Dialog, it is recommended to:

- Set the option `Clean code`
- Unset the option `Enable sections/pragmas/inline/ISR/user attributes`
- Turn off the compute to overflow (CTO) generation. Polyspace can analyze code generated with CTO, but the results may not be as precise.

When you install Polyspace, the `tlcgOptions` variable is updated with `'PolyspaceSupport'`, `'on'` (see variable in `'C:\dSPACE\Matlab\Tl\config\codegen\tl_pre_codegen_hook.m'` file).

See Also

Related Examples

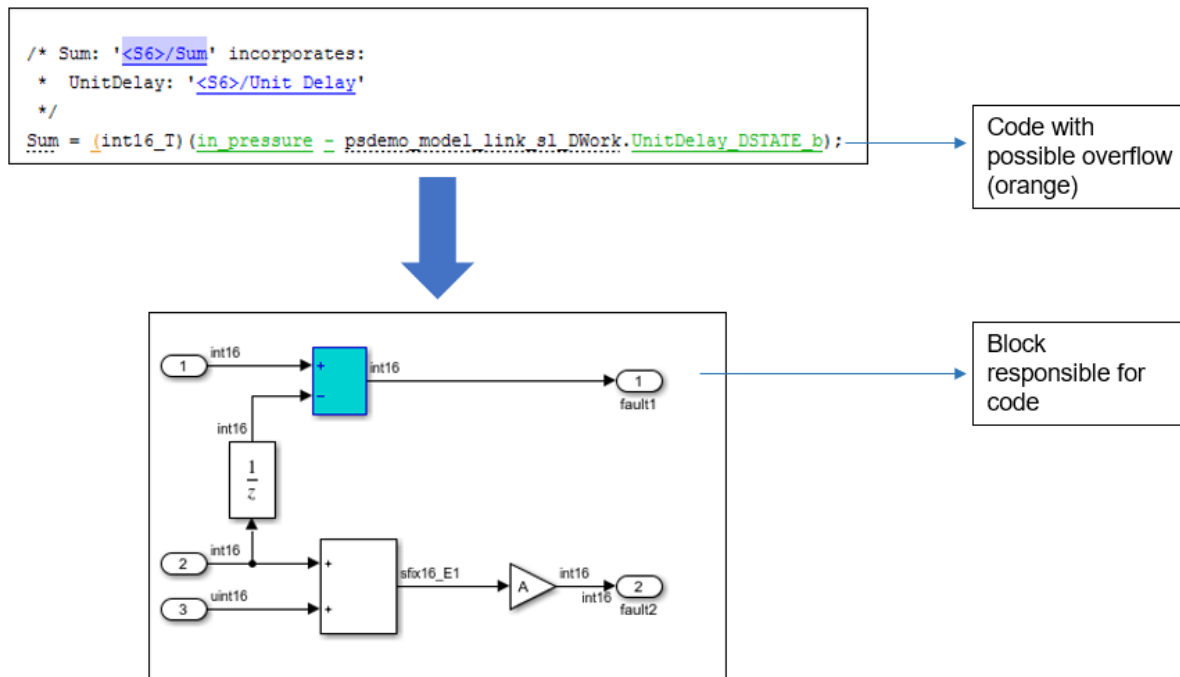
- “Run Polyspace Analysis on Code Generated with TargetLink” on page 6-62

External Websites

- dSPACE - TargetLink

Troubleshoot Navigation from Code to Model

When you run Polyspace on generated code, in the analysis results, you see links in code comments. The links show names of blocks that generate the subsequent lines of code. To see the blocks in the model, you click the block names in the links.



This topic shows the issues that can happen in navigation from code to model.

Links from Code to Model Do Not Appear

See if you are looking at source files (.c or .cpp) or header files. Header files are not directly associated with blocks in the model and do not have links back to the model.

Links from Code to Model Do Not Work

You may encounter issues with the back-to-model feature if:

- Your operating system is Windows Vista® or Windows 7; and User Account Control (UAC) is enabled or you do not have administrator privileges.
- You have multiple versions of MATLAB installed.

To reconnect MATLAB and Polyspace:

- 1 Close Polyspace.
- 2 At the MATLAB command-line, enter `pslinkfun('enablebacktomodel')`.

When you open your Polyspace results, the hyper-links will highlight the relevant blocks in your model.

Your Model Already Uses Highlighting

If your model extensively uses block coloring, the coloring from this feature may interfere with the colors already in your model. You can change the color of blocks when they are linked to Polyspace results. For instance, to change the color to magenta, use this command:

```
color = 'magenta';  
HILITE_DATA = struct('HiliteType', 'find', 'ForegroundColor', 'black', ...  
    'BackgroundColor', color);  
set_param(0, 'HiliteAncestorsData', HILITE_DATA)
```

The color can be one of the following:

- 'cyan'
- 'magenta'
- 'orange'
- 'lightBlue'
- 'red'
- 'green'
- 'blue'
- 'darkGreen'

Polyspace Support of MATLAB and Simulink from Different Releases

Polyspace support of MATLAB or Simulink varies depending on their respective releases. Polyspace fully supports MATLAB and Simulink from the same release, offering complete integration with these software. Polyspace supports MATLAB and Simulink from earlier releases with cross-release integration. See the table.

	Polyspace Release R2018a	Polyspace Release R2018b	Polyspace Release R2019a	Polyspace Release R2019b	Polyspace Release R2020a	Polyspace Release R2020b	Polyspace Release R2021a	Polyspace Release R2021b	Polyspace Release R2022a	Polyspace Release R2022b	Polyspace Release R2023a
MATLAB or Simulink Release R2018a	Complete Integration on page 6-70	"Partial Integration" on page 6-71	"Partial Integration" on page 6-71	"Partial Integration" on page 6-71	"Partial Integration" on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71
MATLAB or Simulink Release R2018b	* on page 6-71	Complete Integration on page 6-70	"Partial Integration" on page 6-71	"Partial Integration" on page 6-71	"Partial Integration" on page 6-71	"Partial Integration" on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71
MATLAB or Simulink Release R2019a	* on page 6-71	* on page 6-71	Complete Integration on page 6-70	"Partial Integration" on page 6-71	"Partial Integration" on page 6-71	"Partial Integration" on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71
MATLAB or Simulink Release R2019b	* on page 6-71	* on page 6-71	* on page 6-71	Complete Integration on page 6-70	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71

MATLAB or Simulink Release R2020a	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	Complete Integration on page 6-70	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71
MATLAB or Simulink Release R2020b	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	Complete Integration on page 6-70	"Cross - Release Integration" on page 6-70	"Cross - Release Integration" on page 6-70	"Cross - Release Integration" on page 6-70	"Cross - Release Integration" on page 6-70	"Cross - Release Integration" on page 6-70
MATLAB or Simulink Release R2021a	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	Complete Integration on page 6-70	"Cross - Release Integration" on page 6-70	"Cross - Release Integration" on page 6-70	"Cross - Release Integration" on page 6-70	"Cross - Release Integration" on page 6-70
MATLAB or Simulink Release R2021b	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	Complete Integration on page 6-70	"Cross - Release Integration" on page 6-70	"Cross - Release Integration" on page 6-70	"Cross - Release Integration" on page 6-70
MATLAB or Simulink Release R2022a	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	Complete Integration on page 6-70	"Cross - Release Integration" on page 6-70	"Cross - Release Integration" on page 6-70

MATLAB or Simulink Release R2022b	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	Complete Integration on page 6-70	“Cross-Release Integration” on page 6-70
MATLAB or Simulink Release R2023a	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	Complete Integration on page 6-70

Note The empty cells (*) in the preceding table represent MATLAB and Simulink support without integration. See “Navigate Back to Model” on page 6-71.

Complete Integration

If MATLAB and Polyspace are from the same release, you can integrate them after installation by calling `polyspacesetup`. See “Same Release of Polyspace and MATLAB” on page 5-2.

You can:

- Run a Polyspace analysis from the Simulink Editor or from the MATLAB Command Window on C/C++ code that is generated from a model or included as custom code in a model. Annotate Simulink blocks and navigate back-to-model from the Polyspace user interface.

See “Bug Finder Analysis in Simulink”.

- Run a Polyspace analysis on C/C++ code that is generated from MATLAB code by using the MATLAB Coder App (if you have Embedded Coder).

See “Bug Finder Analysis in MATLAB Coder”.

- Run a Polyspace analysis on handwritten C/C++ code by using MATLAB scripts.

See “Bug Finder Analysis with MATLAB Scripts”.

Cross-Release Integration

You can integrate Polyspace with MATLAB or Simulink from a release after R2020b. See “MATLAB Release Earlier Than Polyspace” on page 5-3.

This cross-release integration offers limited functionalities. In a cross-release workflow, you can:

- To run a Polyspace analysis on C/C++ code generated by using Embedded Coder, in the MATLAB Command Window, call these functions:

- `pslinkrunCrossRelease`
- `pslinkfun`
- `pslinkoptions`
- Navigate back to your Simulink model from the Polyspace user interface.

You cannot:

- Start a Polyspace analysis of generated code from the Simulink Editor or MATLAB Coder App.
- Start a Polyspace analysis of the custom code included in models or handwritten C/C++ code in the MATLAB Command Window.
- Start a Polyspace analysis of C/C++ code generated from MATLAB code in the MATLAB Command Window.

See “Run Polyspace on Code Generated by Using Previous Releases of Simulink” on page 6-12.

Partial Integration

You can partially integrate Polyspace with MATLAB or Simulink from a release earlier than R2020b. See “MATLAB Release Earlier Than Polyspace” on page 5-3.

This cross-release integration offers limited functionalities. In a cross-release workflow, you can:

- To run a Polyspace analysis on C/C++ code generated by using Embedded Coder, in the MATLAB Command Window, call these functions:
 - `pslinkrun`
 - `pslinkfun`
 - `pslinkoptions`
- Navigate back to your Simulink model from the Polyspace user interface.

You cannot:

- Start a Polyspace analysis of generated code from the Simulink Editor or MATLAB Coder App.
- Start a Polyspace analysis of the custom code included in models or handwritten C/C++ code in the MATLAB Command Window.
- Start a Polyspace analysis of C/C++ code generated from MATLAB code in the MATLAB Command Window.

Navigate Back to Model

You can navigate back to your Simulink model from the Polyspace user interface without integrating Polyspace into your MATLAB or Simulink. Polyspace does not integrate with MATLAB and Simulink if:

- Your MATLAB or Simulink is from a more recent release than your Polyspace.
- Your MATLAB or Simulink is more than four releases behind your Polyspace.

Some specific releases of MATLAB or Simulink do not integrate with Polyspace. See the table.

To navigate back to your model from the Polyspace user interface without integrating Polyspace and MATLAB or Simulink:

- Identify the comments in your code that act as links to the Simulink model. In the **Tools > Preferences > Miscellaneous** tab, select your code generation tool from the context menu **Code comments that act as code-to-model links**. Polyspace recognizes Embedded Coder, MATLAB Coder, and TargetLink. If you use a different code generating tool, select **User Defined**. In the field **Comments beginning with**, specify prefixes of the code comments that act as links.
- In the **Source** pane of the Polyspace user interface, click the code comments that appear as hyperlinks.

See Also

`polyspacesetup` | `pslinkrunCrossRelease`

More About

- “Integrate Polyspace with MATLAB and Simulink” on page 5-2
- “Run Polyspace on Code Generated by Using Previous Releases of Simulink” on page 6-12
- “Fix Issues When when Integrating Polyspace with MATLAB and Simulink” on page 32-65

Run Polyspace Analysis in MATLAB Coder

Run Polyspace on C/C++ Code Generated from MATLAB Code

After generating C/C++ code from MATLAB code, you can independently check the generated code for:

- Bugs or defects and coding rule violations: Use Polyspace Bug Finder.
- Run-time errors: Use Polyspace Code Prover.

Whether you generate code in the MATLAB Coder app or use `codegen`, you can follow the same workflow for checking the generated code.

This tutorial uses the MATLAB Coder example `averaging_filter` in `polyspaceroot\help\toolbox\codeprover\examples\matlab_coder`. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2023a`. The example shows a Code Prover analysis. You can follow a similar workflow for Bug Finder.

Prerequisites

To run this tutorial:

- You must have an Embedded Coder license. The MATLAB Coder app does not show options for running Polyspace unless you have an Embedded Coder license.
- You must be familiar with how to open and use the MATLAB Coder app or the `codegen` command. Otherwise, see the MATLAB Coder Getting Started.
- You must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

Run Polyspace Analysis

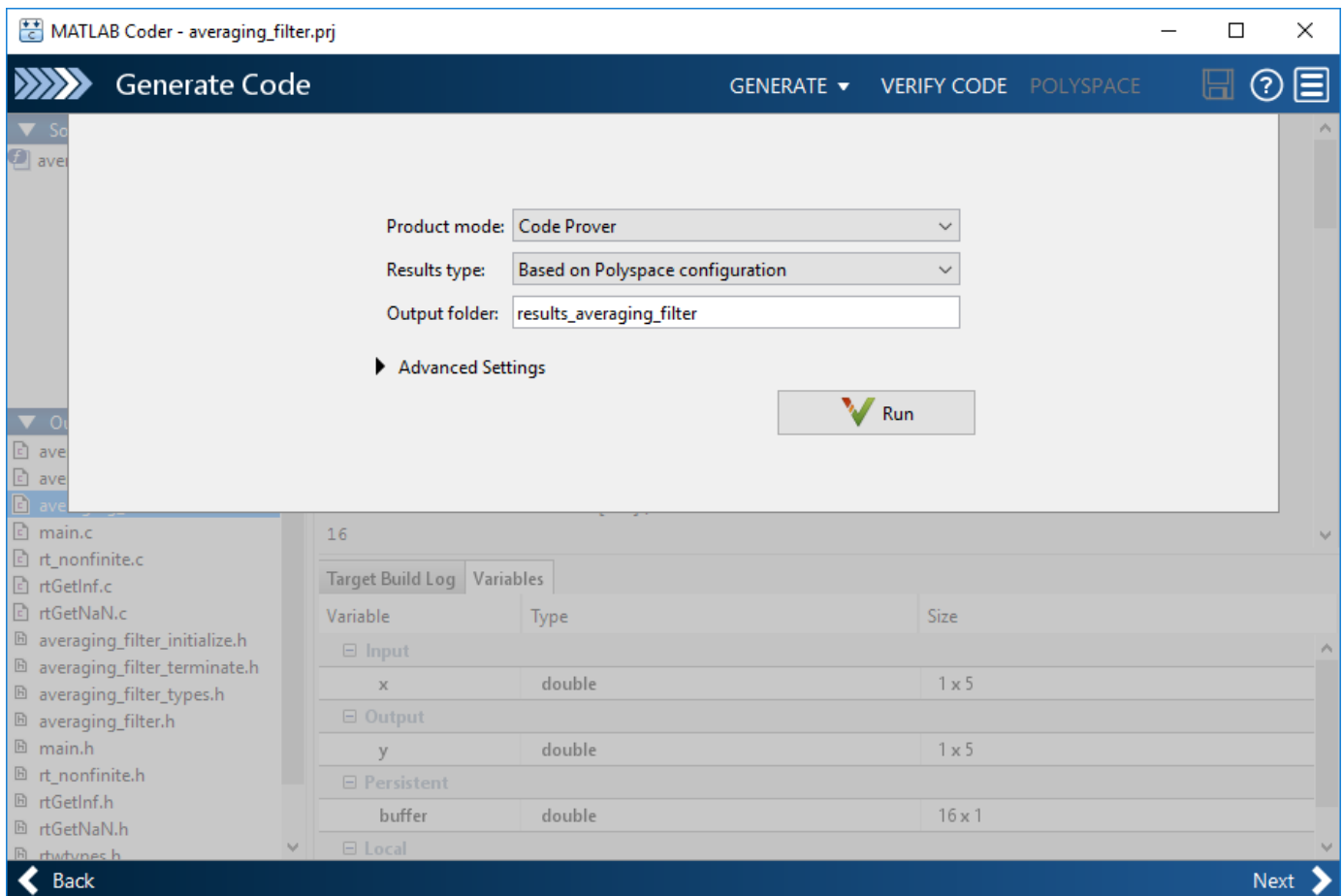
In the MATLAB Coder app, generate code from the file `averaging_filter.m` and analyze the generated code.

- 1 Generate code.

From the entry-point function in the file, generate standalone C/C++ code (a static library, dynamically linked library, or executable program) in the MATLAB Coder app. The function has one input. Explicitly specify a data type for the input, for instance, a 1 X 100 vector of type `double`, or provide a file for deriving data types.

- 2 Analyze the generated code.

After code generation, open the **Polyspace** pane and click **Run**.



If the analysis is completed without errors, the Polyspace results open automatically. If you close the results, you can reopen them from the final page in the app, under the section **Generated Output**. The results are stored in a subfolder `results_averaging_filter` in the folder containing the MATLAB file.

To script the preceding workflow, run:

```
% Generate code
matlabFileName = fullfile(polyspaceroot, 'help',...
    'toolbox','codeprover','examples','matlab_coder','averaging_filter.m');
codegenFolder = fullfile(pwd, 'codegenFolder');
codegen(matlabFileName, '-config:lib', '-c', '-args', ...
    {zeros(1,100,'double')}, '-d', codegenFolder);

% Configure Polyspace analysis
opts = pslinkoptions('ec');
opts.ResultDir = [tempdir 'results'];
opts.OpenProjectManager = 1;

% Run Polyspace
[polyspaceFolder, resultsFolder] = pslinkrun('-codegenfolder', codegenFolder, opts);
```

Review Analysis Results

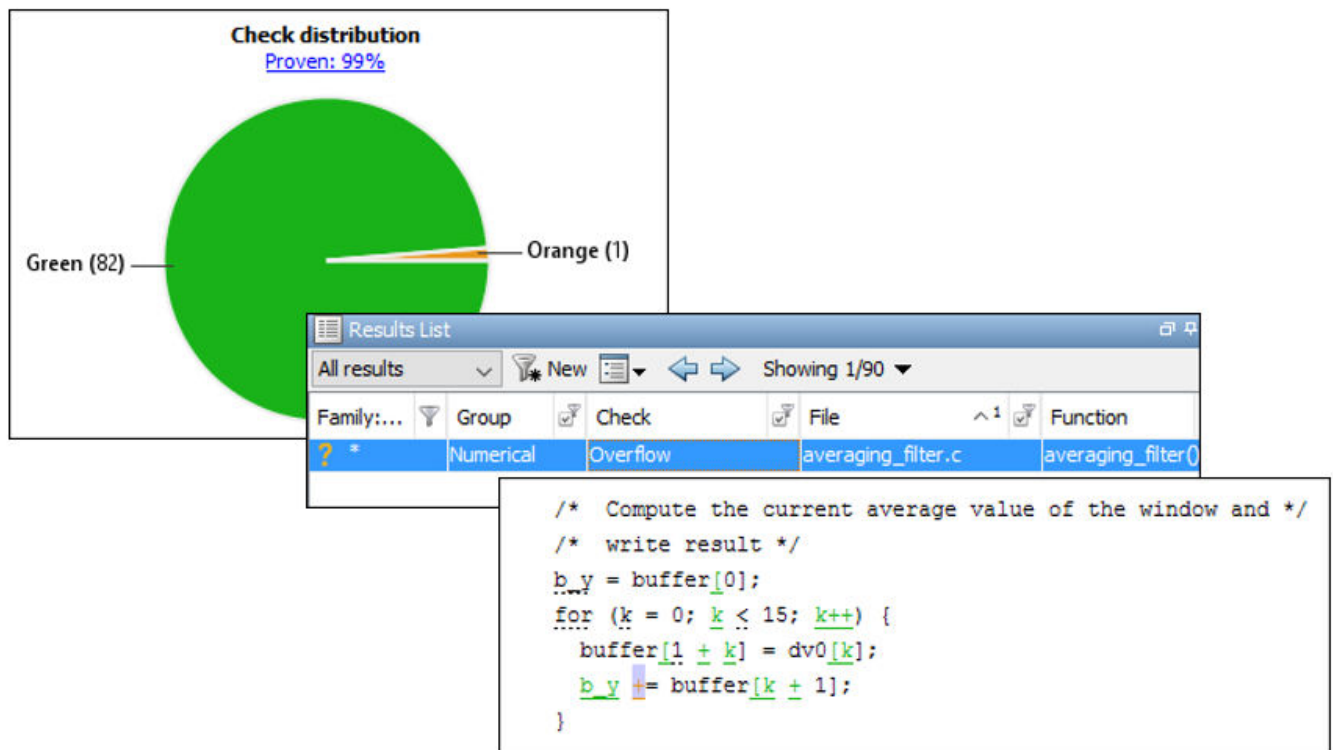
After analysis, the **Results List** pane shows a list of run-time checks. For an explanation of the result colors, see “Code Prover Result and Source Code Colors” (Polyspace Code Prover).

Review the results and determine whether to fix the issues.

- 1 Filter out results that you do not want to review. For instance, you might not want to see the green checks.

See an overview of the results on the **Dashboard** pane. Click the orange section of the pie chart to filter the list of results on the **Results List** pane to the one orange check. Click this orange **Overflow** check and see the source code for the operation that can overflow.

If results are grouped by family, to see a flat list, on the **Results List** pane, from the  dropdown, select **None**.



Check distribution
Proven: 99%

Green (82) — Orange (1)

Family:...	Group	Check	File	Function
? *	Numerical	Overflow	averaging_filter.c	averaging_filter()

```

/* Compute the current average value of the window and */
/* write result */
b_y = buffer[0];
for (k = 0; k <= 15; k++) {
    buffer[1 + k] = dv0[k];
    b_y += buffer[k + 1];
}

```

- 2 Find the root cause of each run-time error.

On the **Source** pane, use right-click navigation tools and tooltips to identify the root cause of the check. In this case, you see that the + operation overflows because Polyspace makes an assumption about the input array to the function. The assumption is that the array elements can have any value allowed by their double data type. The tooltip on the line `buffer[0] = x[i]` shows the assumed range.


```

/* Add a new sample value to the buffer */
buffer[0] = x[i];

/* Com Assignment to element of static array (float 64): [-1.7977E+308 .. 1.7977E+308]
/* wri
b_y = b array size: 16
for (k array index value: 0
  buffe
  b_y += buffer[k + 1];
}

```

With an Embedded Coder license, you can easily trace back from the generated C code to the original MATLAB code. See “Interactively Trace Between MATLAB Code and Generated C/C++ Code” (Embedded Coder).

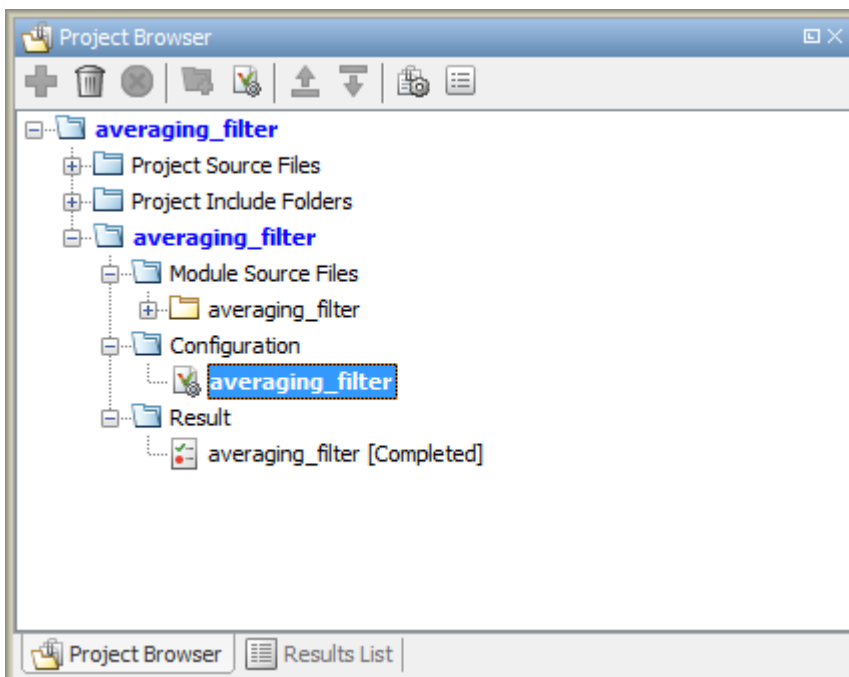
Run Analysis for Specific Design Range

You can check the generated code for a specific range of inputs. Range specification helps narrow down the default assumption that inputs are full-range.

To specify a range for inputs:

- 1 Open the analysis configuration.

In the Polyspace user interface, switch to the Polyspace project created for the analysis. Select **Window > Reset Layout > Project Setup**. On the **Project Browser** pane, click the project configuration.



- Specify a design range for the inputs.

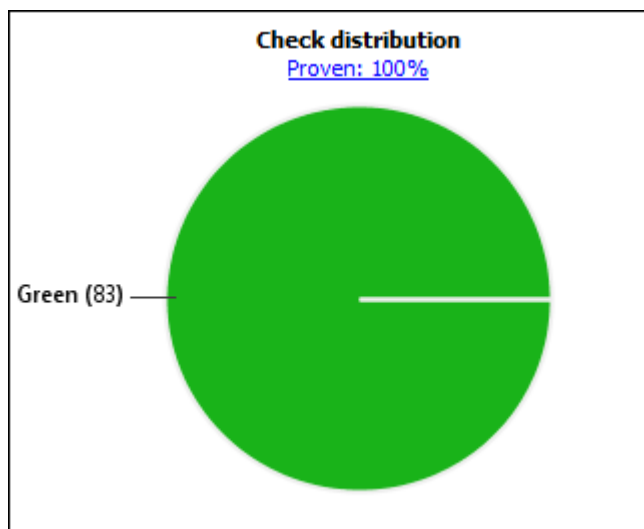
In the **Configuration** pane, on the **Inputs & Stubbing** node, set up your constraints. Click **Edit** beside **Constraint setup**. Constrain the range of the first input to [-100..100].

Name	File	Main Generator Called	Init Mode	Init Range
Global Variables				
[-] User Defined Functions				
[-] averaging_filter()	averaging_filter.c	MAIN GENERATOR		
[-] averaging_filter.arg1	averaging_filter.c		INIT	
[-] averaging_filter.* arg1	averaging_filter.c		INIT	-100..100
[+] averaging_filter.arg2	averaging_filter.c		INIT	

You can overwrite the default constraint template or save the constraints elsewhere. For information on the columns in this window, see “External Constraints for Polyspace Analysis” on page 14-6.

- Rerun the analysis from the Coder app (or at the MATLAB command line) and see the results.

On the **Dashboard** pane, you do not see the previous orange overflow anymore.



See Also

pslinkrun

More About

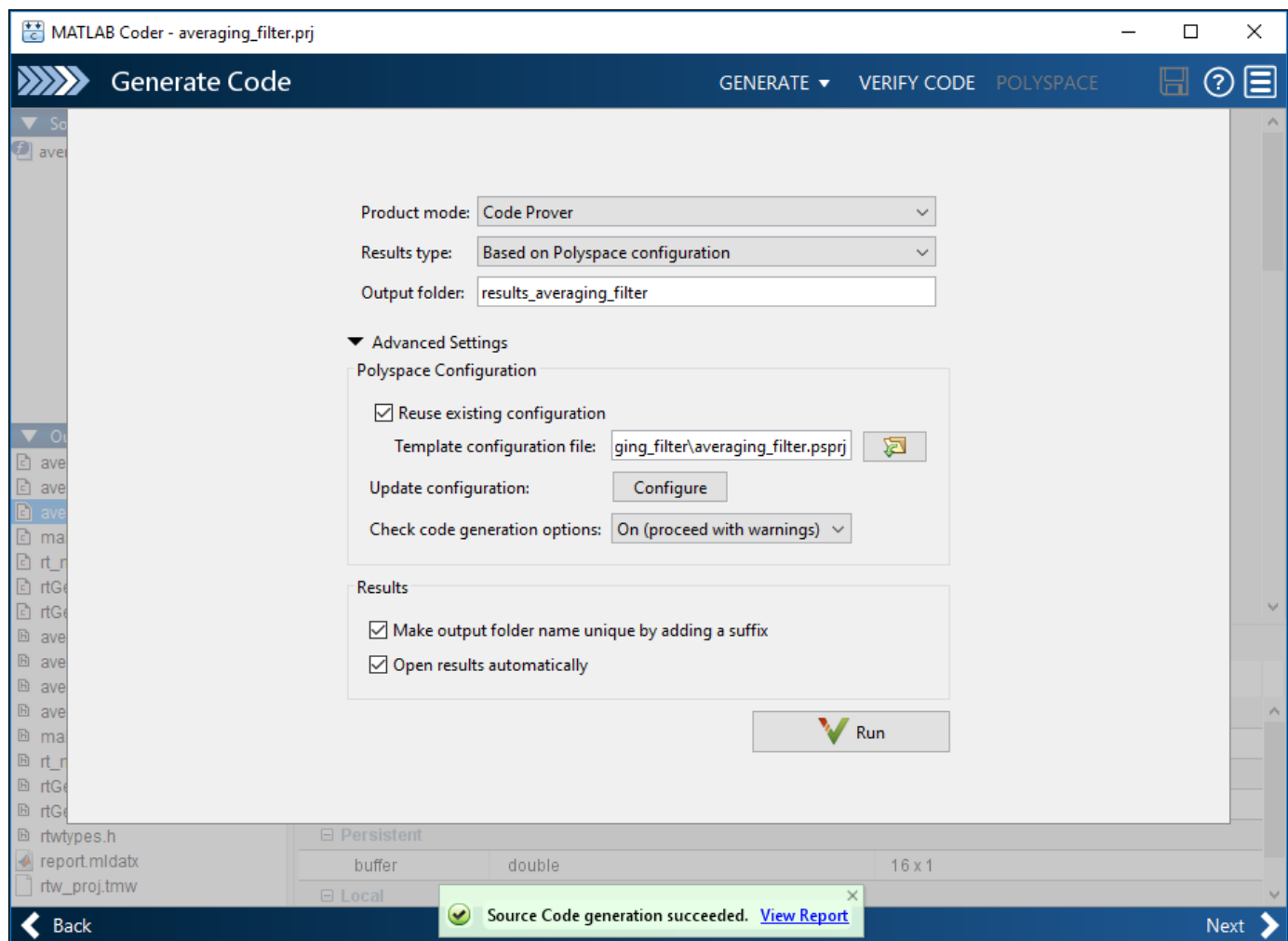
- “Configure Advanced Polyspace Options in MATLAB Coder App” on page 7-7

Configure Advanced Polyspace Options in MATLAB Coder App

Before analyzing generated code with Polyspace in the MATLAB Coder App, you can change some of the default options. This topic shows how to configure the options and save this configuration.

For getting started with Polyspace analysis in the MATLAB Coder App, see “Run Polyspace on C/C++ Code Generated from MATLAB Code” on page 7-2.

Configure Options



The default analysis runs Code Prover based on a default project configuration. The results are stored in a folder `result_project_name` in the current working folder.

You can change these options in the MATLAB Coder App itself:

- **Product mode:** Select Code Prover or Bug Finder.
- **Results type:** Check for MISRA C:2004 (MISRA AC AGC) or MISRA C:2012 rule violations, in addition to or instead of the default checkers.

- **Output folder:** Choose an output folder name. To save the results of each run in a new folder, under **Advanced Settings**, select **Make output folder name unique by adding a suffix**.
- **Check code generation options:** Choose to see warnings or errors if the code generation uses options that can result in imprecise Code Prover analysis.

For instance, if the code generation setting **Use memset to initialize floats and doubles to 0.0** is disabled, Code Prover can show imprecise orange checks because of approximations. See “Orange Checks in Polyspace Code Prover” (Polyspace Code Prover).

To see the other default options or update them, under **Advanced Settings**, click the **Configure** button. You see the options on a **Configuration** pane.

For more information on the options, see Bug Finder Analysis Options or Code Prover Analysis Options (Polyspace Code Prover).

Share and Reuse Configuration

If you change some of the default options in the **Configuration** pane, your updated configuration is saved as a `.psprj` file in the results folder. Using this file, you can reuse your configuration across multiple MATLAB Coder projects.

Reuse Configuration in Coder App

To reuse a previous configuration in the current project opened in the MATLAB Coder App, under **Advanced Settings**, select **Reuse existing configuration**. For **Template configuration file**, provide the `.psprj` file that stores the previous configuration.

The **Results type** option in the MATLAB Coder app still shows **Based on Polyspace configuration** but the configuration used is the one that you provided.

Reuse Configuration on Command Line

At the MATLAB command line, you create an options object with the `pslinkoptions` function. You modify the analysis options by using the properties of this object and then run analysis with the `pslinkrun` function.

```
opts = pslinkoptions('ec');  
...  
pslinkrun('-codegenfolder', codegenFolder, opts);
```

You can associate advanced analysis options set in a `.psprj` file with the options object. Use the properties `EnablePrjConfigFile` and `PrjConfigFile`.

```
opts.EnablePrjConfigFile = true;  
opts.PrjConfigFile = 'C:\Polyspace\config.psprj';
```

For more information, see `pslinkoptions` Properties.

See Also

`pslinkoptions`

More About

- “Run Polyspace on C/C++ Code Generated from MATLAB Code” on page 7-2

Configure Analysis on Servers

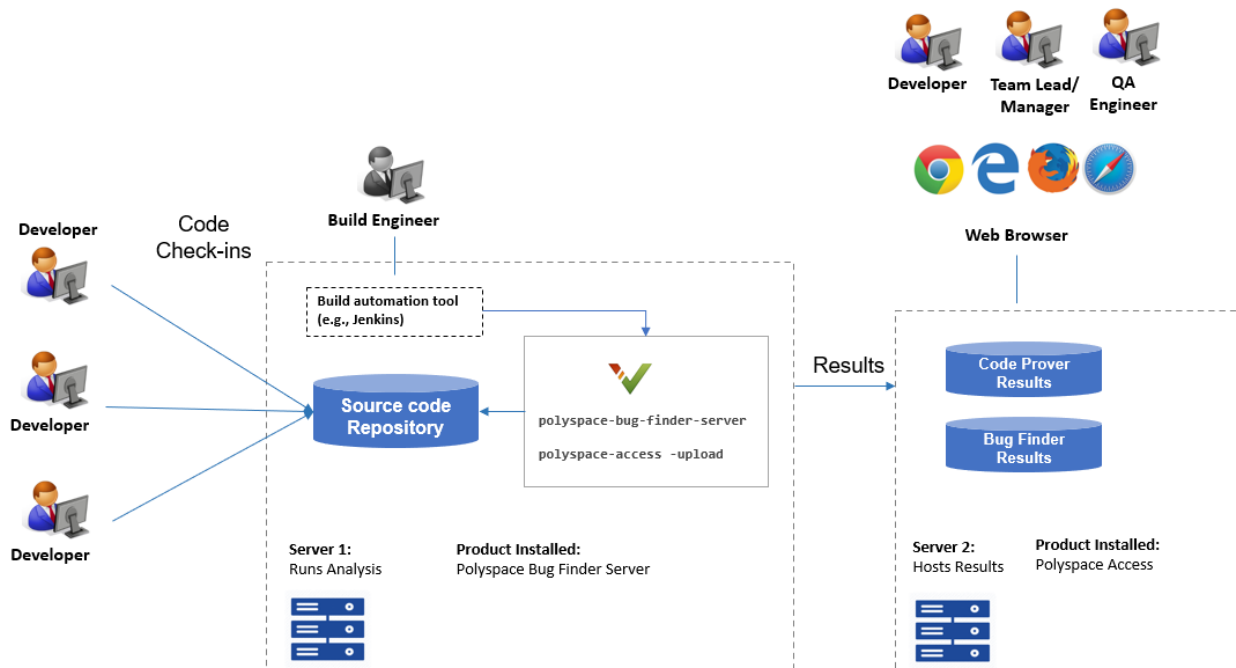
Run Polyspace Analysis on Servers

Run Polyspace Bug Finder on Server and Upload Results to Polyspace Access Web Interface

Polyspace Bug Finder Server checks C/C++ code for defects and coding standard violations, and then uploads findings to a web interface for code review.

You can run Bug Finder as part of continuous integration. Set up scripts that run a Bug Finder analysis at regular intervals or based on new code submissions. The scripts can upload the analysis results for review in the Polyspace Access web interface and optionally send emails to owners of source files with Polyspace findings. The owners can open the web interface to review only the new findings from their submission, and then fix or justify the issues.

In a typical project or team, Polyspace Bug Finder Server runs periodically on a few testing servers and uploads the results for review. Each developer and quality engineer in the team has a Polyspace Bug Finder Access license to view the results in the web interface for investigation and bug fixing.



Note: Depending on the specifications, the same computer can serve as both Server 1 and Server 2.

Prerequisites

To run a Bug Finder analysis on a server and review the results in the Polyspace Access web interface, perform this one-time setup:

- To run the analysis, install one instance of the Polyspace Bug Finder Server product.
- To upload results, set up the components required to host the web interface of Polyspace Access.
- To view the uploaded results, you and each developer reviewing the results must have a Polyspace Bug Finder Access license.

See “Install Polyspace Server and Access Products”.

Check Polyspace Installation

To check if Polyspace Bug Finder Server is installed:

- 1 Open a command window. Navigate to *polyspaceserverroot*\polyspace\bin. Here, *polyspaceserverroot* is the Polyspace Bug Finder Server installation folder, for instance, C:\Program Files\Polyspace Server\R2023a. See also “Installation Folder”.
- 2 Enter:

```
polyspace-bug-finder-server -help
```

You should see the list of options allowed for a Bug Finder analysis.

To check if the Polyspace Access web interface is set up for upload:

- 1 Navigate again to *polyspaceserverroot*\polyspace\bin.
- 2 Enter:

```
polyspace-access -host hostName -port portNumber -create-project testProject
```

Here, *hostName* is the name of the server hosting the Polyspace Bug Finder Access web server. For a locally hosted server, use `localhost`. The *portNumber* is the optional port number of the server. If you omit the port number, 9443 is used.

If the setup was complete, a project called `testProject` is created in the Polyspace Access web interface.

You are prompted for your login and password each time that you use the `polyspace-access` command. To avoid entering login information each time, provide the login and an encrypted version of your password with the command. To create an encrypted password, enter:

```
polyspace-access -encrypt-password
```

Enter your login and password. Copy the encrypted password and provide this encrypted password with the `-encrypted-password` option when using the `polyspace-access` command.

- 3 In a web browser, open this URL:

```
https://hostName:portNumber/metrics/index.html
```

Here, *hostName* and *portNumber* are the host name and port number from the previous step.

In the **Project Explorer** pane on the Polyspace Access web interface, you see the newly created project `testProject`.

Run Bug Finder on Sample Files

To run Bug Finder, in your operating system, open a command window.

- 1 To run a Bug Finder analysis, use the `polyspace-bug-finder-server` command.
- 2 To upload the results to the Polyspace Access web interface, use the `polyspace-access` command.

To avoid typing the full path to the command, add the path `polyspaceserverroot\polyspace\bin` to the Path environment variable on your operating system.

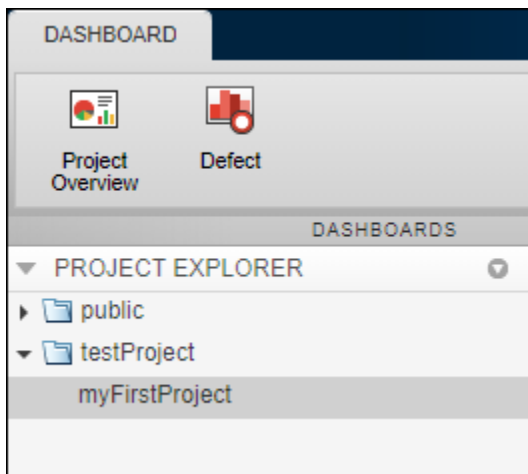
Try out the commands on sample files provided with your Polyspace installation.

- 1 Copy the sample source files from `polyspaceserverroot\polyspace\examples\cxx\Bug_Finder_Example\sources` to another folder where you have write permissions. Navigate to this folder at the command line.
- 2 Enter:

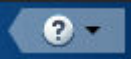
```
polyspace-bug-finder-server -sources numerical.c,dataflow.c -I .
                           -checkers numerical,data_flow -results-dir .
polyspace-access -host hostName -port portNumber
                 -login username -encrypted-password pwd
                 -create-project testProject
polyspace-access -host hostName -port portNumber
                 -login username -encrypted-password pwd
                 -upload . -project myFirstProject -parent-project testProject
```

Here, *username* is your login name and *pwd* is the encrypted password that you created previously. See “Check Polyspace Installation” on page 8-3.

Refresh the Polyspace Access web interface. You see a folder `testProject` on the **Project Explorer** pane. The folder contains one project `myFirstProject`.



To see the results in the project, click **Review**. For more information, see “Review Polyspace Bug

Finder Results in Web Browser”. You can also access the documentation using the  button in the upper right of the Polyspace Access interface.

The analysis options used with the `polyspace-bug-finder-server` command are:

- `-sources`: Specify comma-separated source files.
- `-I`: Specify path to include folder. Use the `-I` flag each time you want to add a separate include folder.
- Find defects (`-checkers`): Specify the defects (bugs) to check for.

- `-results-dir`: Specify the path to the folder where Polyspace Bug Finder results will be saved.

Note that the results folder is cleaned up and repopulated at each run. To avoid accidental removal of files during the cleanup, instead of using an existing folder that contains other files, specify a dedicated folder for the Polyspace results.

For the full list of options available for a Bug Finder analysis, see “Complete List of Polyspace Bug Finder Analysis Engine Options”. To open the Bug Finder documentation in a help browser, enter:

```
polyspace-bug-finder-server -doc
```

Sample Scripts for Bug Finder Analysis on Servers

To run the analysis, instead of typing the commands at the command line, you can use scripts. The scripts can execute each time that you add or modify source files.

A sample Windows batch file is shown below. Here, the path to the Polyspace installation is specified in the script. To use this script, replace `polyspaceserverroot` with the path to your installation. You must have already generated the encrypted password for use in the scripts. See “Check Polyspace Installation” on page 8-3.

```
echo off
set POLYSPACE_PATH=polyspaceserverroot\polyspace\bin
set LOGIN=-host hostName -port portNumber -login username -encrypted-password pwd
"%POLYSPACE_PATH%\polyspace-bug-finder-server" -sources numerical.c,dataflow.c -I . ^
-checkers numerical,data_flow -results-dir .
"%POLYSPACE_PATH%\polyspace-access" %LOGIN% -create-project testProject
"%POLYSPACE_PATH%\polyspace-access" %LOGIN% -upload . -project myFirstProject
-parent-project testProject
pause
```

A sample Linux shell script is shown below.

```
POLYSPACE_PATH=polyspaceserverroot/polyspace/bin
LOGIN=-host hostName -port portNumber -login username -encrypted-password pwd
${POLYSPACE_PATH}/polyspace-bug-finder-server -sources numerical.c,dataflow.c -I . \
-checkers numerical,data_flow -results-dir .
${POLYSPACE_PATH}/polyspace-access $LOGIN -create-project testProject
${POLYSPACE_PATH}/polyspace-access $LOGIN -upload . -project myFirstProject
-parent-project testProject
```

Specify Sources and Options in Separate Files from Launching Scripts

Instead of listing the source files and analysis options within the launching scripts, you can list them in separate text files.

- Specify the text file listing the sources by using the option `-sources-list-file`.
- Specify the text file listing the analysis options by using the option `-options-file`.

By removing the source files and analysis option specifications from the launching scripts, you can modify these specifications as required with new code submissions while leaving the launching script untouched.

Consider the script in the preceding example. You can modify the `polyspace-bug-finder-server` command to use text files with sources and options. Instead of:

```
polyspace-bug-finder-server -sources numerical.c,dataflow.c
                             -I . -checkers numerical,data_flow -results-dir .
```

use:

```
polyspace-bug-finder-server -sources numerical.c,dataflow.c
                             -I . -checkers numerical,data_flow -results-dir .
```

Here:

- `sources.txt` lists the source files in separate lines:

```
numerical.c
dataflow.c
```

- `polyspace_opts.txt` lists the analysis options in separate lines:

```
-I .
-checkers numerical,data_flow
```

Typically, your source files are specified in a build command (makefile). Instead of specifying the source files directly, you can trace the build command to create a list of source specifications. See `polyspace-configure`.

Complete Workflow

In a typical continuous integration workflow, you run a script that executes these steps:

- 1 Extract Polyspace options from your build command.

For instance, if you use makefiles to build your source code, you can extract analysis options from the makefile.

```
polyspace-configure -output-options-file compile_opts make
```

See also:

- `polyspace-configure`
 - “Create Polyspace Analysis Configuration from Build Command (Makefile)” on page 13-22
- 2 Run the analysis with the previously created options file. Append a second options file that contains the remaining options required for the analysis.

```
polyspace-bug-finder-server -options-file compile_opts -options-file run_opts
```

- 3 Upload the results to Polyspace Bug Finder Access.

```
polyspace-access login -upload resultsFolder -project projName
                    -parent-project parentProjName
```

Here, `login` is the combination of options required to communicate with the web server that is hosting Polyspace Bug Finder Access:

```
-host hostName -port portNumber -login username -encrypted-password pwd
```

resultsFolder is the folder containing the Polyspace results. *projName* and *parentProjName* are names of the project and parent folder as they would appear in the Polyspace Access web interface.

- 4 Optionally, send email notifications to developers with new results from their code submission. The email contains attachments with links to the results in the Polyspace Access web interface.

See “Send Email Notifications with Polyspace Bug Finder Server Results”.

See examples of scripts executing these steps in “Sample Scripts for Polyspace Analysis with Jenkins” on page 8-17.

See Also

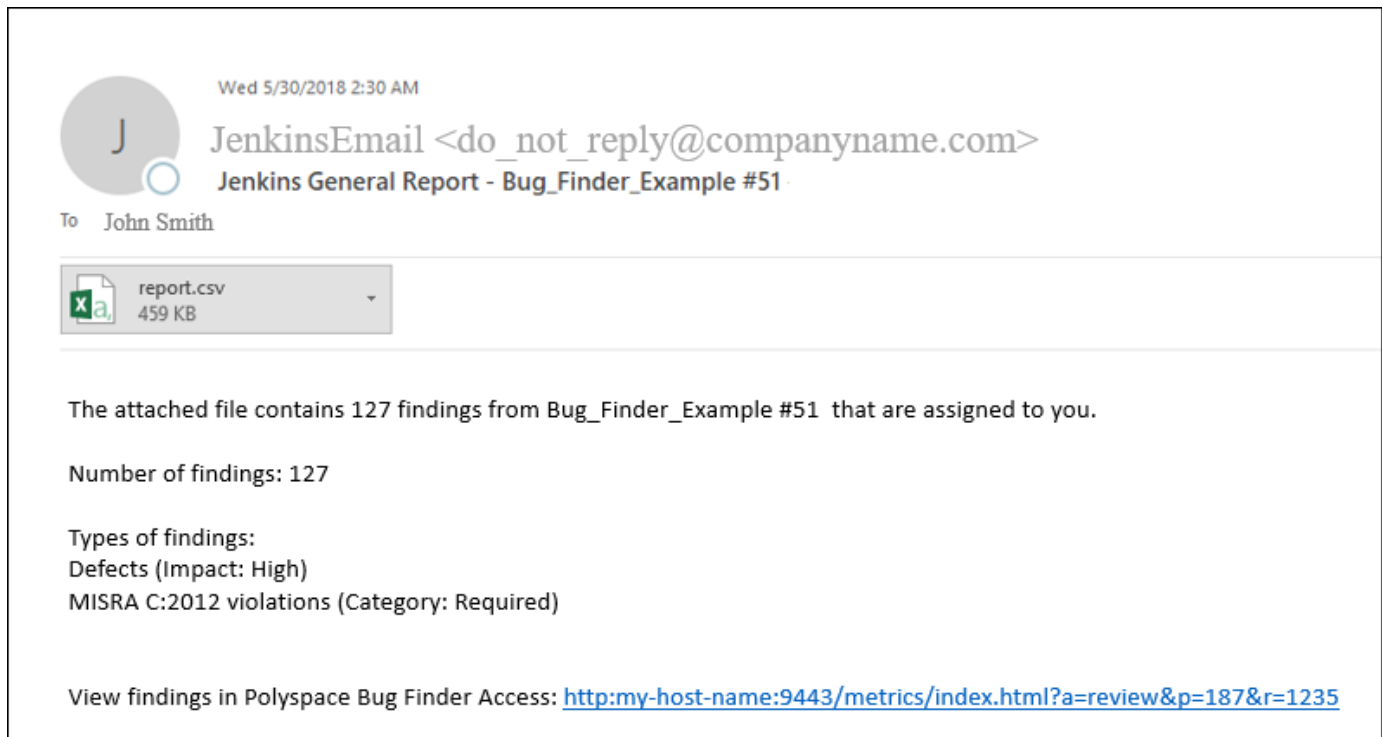
polyspace-access | polyspace-bug-finder-server

More About

- “Send Email Notifications with Polyspace Bug Finder Server Results”
- “Complete List of Polyspace Bug Finder Analysis Engine Options”

Send Email Notifications with Polyspace Bug Finder Server Results

If you run a Polyspace analysis as part of continuous integration, each new code submission produces new results. You not only see new results in components that were modified but also in components that depended on the modified components. You can set up e-mail alerts so that component owners get notified when new Polyspace results appear in their components.



Creating E-mail Notifications

To create e-mail notifications:

- 1 Export new analysis results to a tab-delimited text file (.tsv format). For each result, the file contains links to open the result in the Polyspace Access web interface.

Apply filters to export specific types of results, for instance, defects with high impact. If required, you can also apply additional filters to the exported files using search and replace utilities. See “Export Results for E-mail Attachments” on page 8-10.

- 2 Send an email with the results file in attachment.

For instance, if you use an e-mail plugin in Jenkins, you can create a post-build step to send an e-mail after the analysis is complete.

If you use the Polyspace plugin in Jenkins, you can use Polyspace helper utilities for the entire e-mail notification process. See “Sample Scripts for Polyspace Analysis with Jenkins” on page 8-17.

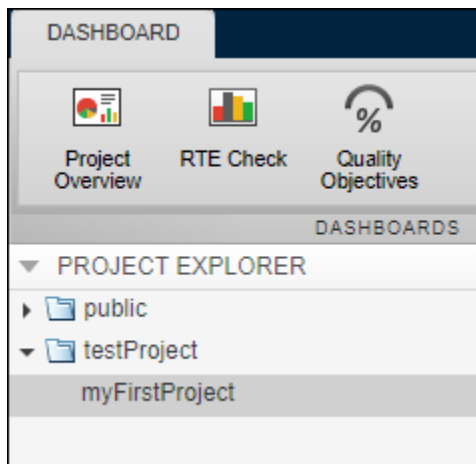
Alternatively, results can be directly assigned to owners based on their file paths. You can set up email notifications that exports a separate results file per owner and sends an email to each owner

with the corresponding results file in attachment. See “Assign Owners and Export Assigned Results” on page 8-10.

Prerequisites

To run this tutorial:

- You must have uploaded some result in the Polyspace Bug Finder Access interface. If you complete the tutorial “Run Polyspace Bug Finder on Server and Upload Results to Web Interface”, you should see a folder `testProject` on the **Project Explorer** pane. The folder contains one project `myFirstProject`.



To see the results in the project, with `myFirstProject` selected, click the **Review** button. You see a list of defects. The **Information** column shows the impact of the defects. In this tutorial, only high-impact defects will be exported for e-mail attachments.

- You must be able to interact with the Polyspace Bug Finder Access interface from the command line. For instance, navigate to `polyspaceserverroot\polyspace\bin` and enter:

```
polyspace-access login -list-project
```

Here, `polyspaceserverroot` is the Polyspace Bug Finder Server installation folder, for instance, `C:\Program Files\Polyspace Server\R2023a`. The variable `login` refers to the following combination of options. You provide these options with every use of the `polyspace-access` command.

```
-host hostName -port portNumber -login username -encrypted-password pwd
```

Here, `hostName` is the name of the Polyspace Bug Finder Access web server. For a locally hosted server, use `localhost`. `portNumber` is the optional port number of the server. If you omit the port number, `9443` is used. `username` and `pwd` refer to the login and an encrypted version of your password. To create an encrypted password, enter:

```
polyspace-access -encrypt-password
```

Copy the encrypted password and provide this password with later uses of the `polyspace-access` command.

Export Results for E-mail Attachments

You can export all results in a project or only certain types of results.

Open a command window. Navigate to the folder where you want to export the results.

- To export all results, enter the following:

```
polyspace-access login -export testProject/myFirstProject -output .\result.txt
```

- To export only defects with high impact, enter the following:

```
polyspace-access login -export testProject/myFirstProject -defects High
-output .\result_high_impact.txt
```

Open each text file in a spreadsheet viewing utility such as Microsoft® Excel®. In the first file, you see all defects but in the second file, you only see the defects with high impact. Instead of `-defects High`, you can apply other filters. For instance:

- To see only new defects compared to the previous analysis of the same project, use the option `-new-findings`.
- To apply a more fine-grained set of filters, you can use software quality objectives (SQOs). The software quality objectives are specified through a progressively stricter set of SQO levels, numbered from 1 to 6. You can customize the requirements of each level in the Polyspace Access web interface, and then use the option `-open-findings-for-sqo` with the level number to export only those results that must be reviewed to meet the requirements. See also “Evaluate Polyspace Bug Finder Results Against Bug Finder Quality Objectives” on page 31-2.

To see all filtering options, enter:

```
polyspace-access -h -export
```

You can configure your e-mail utility to send these exported files in attachment.

If required, you can also apply additional filters to the exported files using search and replace utilities. For instance, use search and replace utilities on the results file to include results only from specific files and functions. In Linux, you can use `grep` and `sed` to retain only results in specific files.

Assign Owners and Export Assigned Results

You can assign owners to results in specific files or folders. You can then export one result file per owner and send an email to each owner with the corresponding file in attachment.

You can assign owners in the Polyspace Access web interface or at the command line.

In this tutorial, assign all results in the file `numerical.c` to `jsmith` and all results in the file `dataflow.c` to `jboyd`.

```
polyspace-access login
-set-unassigned-findings testProject/myFirstProject
-owner jsmith -source-contains numerical.c
polyspace-access login
-set-unassigned-findings testProject/myFirstProject
-owner jboyd -source-contains dataflow.c
```

After assignment, export one results file per owner.

```
polyspace-access login  
-export testProject/myFirstProject -output .\results.txt -output-per-owner
```

These files contain the exported results:

- `results.txt` contains all results.
- `results_jsmith.txt` and `results_jboyd.txt` contains results assigned to `jsmith` and `jboyd` respectively.
- `results.txt.owners.list` contains the list of owners, in this case:

```
jsmith  
jboyd
```

Before assigning owners to results, use the option `-dryrun` to perform a dry run of the assignments. Without performing the assignment, the option shows the files with results that are assigned and the owner that the results are assigned to.

See Also

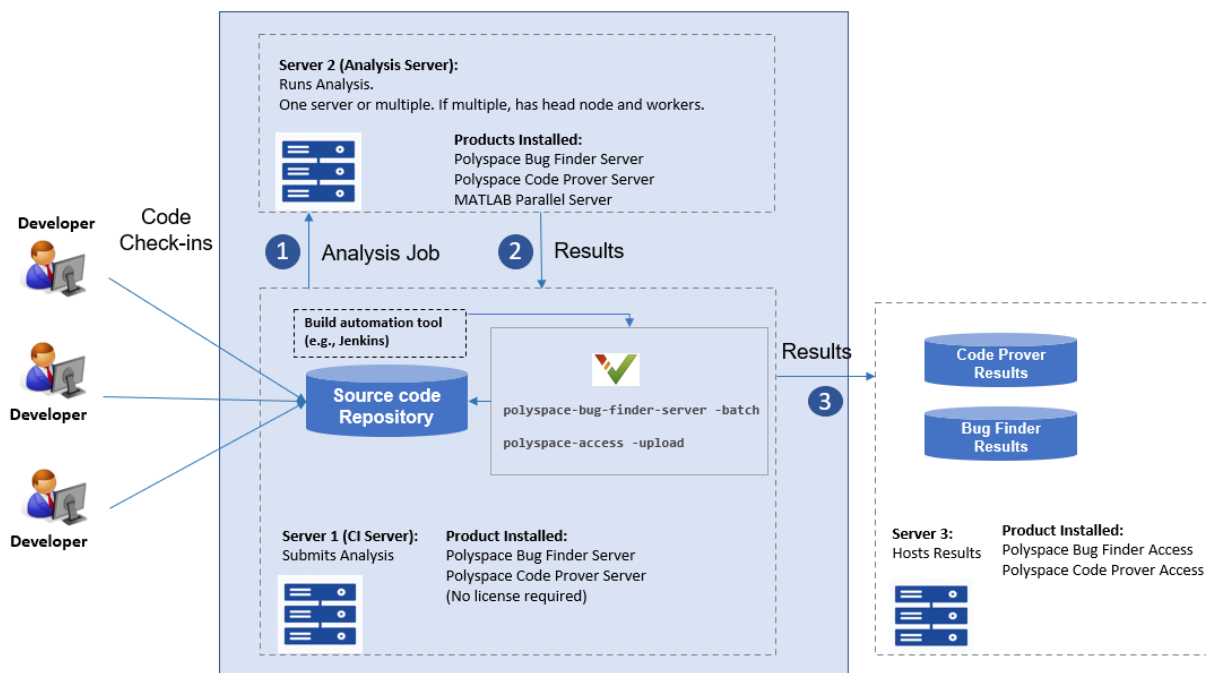
`polyspace-access`

Offload Polyspace Analysis from Continuous Integration Server to Another Server

When running static code analysis with Polyspace as part of continuous integration, you might want the analysis to run on a server that is different from the server running your continuous integration (CI) scripts. For instance:

- You might want to perform the analysis on a server that has more processing power. You can offload the analysis from your CI server to the other server.
- You might want to submit analysis jobs from several CI servers to a dedicated analysis server, hold the jobs in queue, and execute them as Polyspace Server instances become available.

When you offload an analysis, the compilation phase of the analysis runs on the CI server. After compilation, the analysis job is submitted to the other server and continues on this server. On completion, the analysis results are downloaded back to the CI server. You can then upload the results to Polyspace Access for review, or report the results in some other format.



Install Products

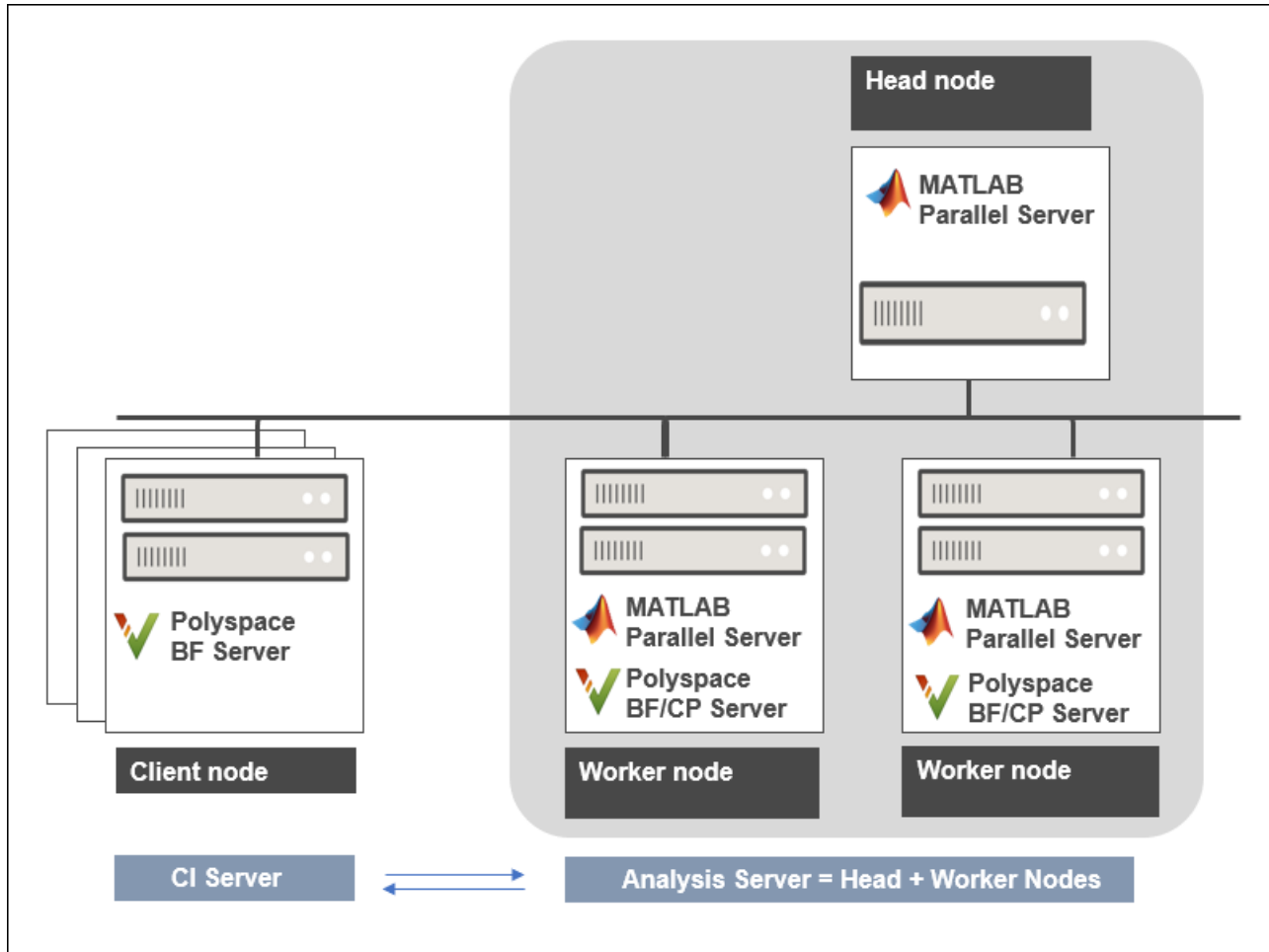
A typical distributed network for offloading an analysis consists of these parts:

- **Client node(s):** Each CI server acts as a client node that submits Polyspace analysis jobs to a cluster.

The cluster consists of a head node and one or more worker nodes. In this example, we use the same computer as the head node and one worker node.

- **Head node:** The head node distributes the submitted jobs to worker nodes.
- **Worker node(s):** Each worker node executes one Polyspace analysis at a time.

Note The versions of Polyspace on the client and worker nodes must match.



Install these products:

- **Client nodes:** Polyspace Bug Finder Server or Polyspace Code Prover Server to submit jobs from the Continuous Integration server. Note that you do not require licenses for the Polyspace Server products if you use them only for job submission (with the `-batch` option).
- **Head node:** MATLAB Parallel Server™ to manage submissions from multiple clients. An analysis job is created for each submission and placed in a queue. As soon as a worker node is available, the next analysis job from the queue is run on the worker.
- **Worker node(s):** MATLAB Parallel Server and Polyspace Bug Finder Server or Polyspace Code Prover Server on the worker nodes to run a Bug Finder or Code Prover analysis.

In the simplest configuration, where the same computer serves as the head node and one worker node, you install MATLAB Parallel Server and one or both Polyspace Bug Finder Server and Polyspace

Code Prover Server on this computer. This example describes the simple configuration but you can generalize the steps to multiple workers on separate computers.

Configure and Start Job Scheduler Services on Head Node and Worker Node

Start a job scheduler service (the MATLAB Job Scheduler or `mjs` service) on the computer that acts as the head node and worker node. Before starting the service, you must perform an initial setup.

Specify Polyspace Installation Paths

MATLAB Parallel Server and Polyspace Server products are installed in two separate folders. The MATLAB Parallel Server installation routes the Polyspace analysis to the Polyspace Server products. To link the two installations, specify the path to the root folder of the Polyspace Server products in your MATLAB Parallel Server installation.

- 1 Navigate to `matlabroot\toolbox\parallel\bin\`. Here, `matlabroot` is the MATLAB Parallel Server installation folder, for instance, `C:\Program Files\MATLAB\R2023a`.
- 2 Uncomment and modify the following line in the file `mjs_polyspace.conf`. To edit and save the file, open your editor in administrator mode.

```
POLYSPACE_SERVER_ROOT=polyspaceserverroot
```

Here, `polyspaceserverroot` is the installation path of the server products, for instance:

```
C:\Program Files\Polyspace Server\R2023a
```

The Polyspace Server product offloading the analysis must belong to the same release as the Polyspace Server product running the analysis. If you offload an analysis from an R2023a Polyspace Server product, the analysis must run using another R2023a Polyspace Server product.

Configure mjs Service Settings

Before starting MATLAB Parallel Server (the `mjs` service), you must perform a minimum configuration.

- 1 Navigate to `matlabroot\toolbox\parallel\bin`, where `matlabroot` is the MATLAB Parallel Server installation folder, for instance, `C:\Program Files\MATLAB\R2023a`.
- 2 Modify the file `mjs_def.bat` (Windows) or `mjs_def.sh` (Linux). To edit and save the file, open your editor in administrator mode.

Read the instructions in the file and uncomment the lines as needed. At a minimum, uncomment these lines that specify:

- Host name.

Windows:

```
REM set HOSTNAME=%strHostname%.%strDomain%
```

Linux:

```
#HOSTNAME=`hostname -f`
```

Explicitly specify your computer host name.

- Security level.

Windows:

```
REM set SECURITY_LEVEL=
```

Linux:

```
#SECURITY_LEVEL=""
```

Explicitly specify a security level to avoid future errors when starting the job scheduler.

For security levels 2 and higher, you have to provide a password in a graphical window at the time of job submission.

Start mjs Service and One Worker

In a command-line terminal, cd to *matlabroot*\toolbox\parallel\bin, where *matlabroot* is the MATLAB Parallel Server installation folder, for instance, C:\Program Files\MATLAB\R2023a. Run these commands (directly at the command line or by using scripts):

```
mjs install
mjs start
startjobmanager -name JobScheduler -remotehost hostname -v
startworker -jobmanagerhost hostname -jobmanager JobScheduler
               -remotehost hostname -v
```

Here, *hostname* is the host name of your computer. This name is the host name that you specified in the file *mjs_def.bat* (Windows) or *mjs_def.sh* (Linux).

For more details and configuring services with multiple workers, see:

- “Install and Configure MATLAB Parallel Server for MATLAB Job Scheduler and Network License Manager” (MATLAB Parallel Server)
- *mjs*

Offload Analysis from Client Node

Once you have set up the computer that acts as the head node and worker node, you are ready to offload a Polyspace analysis from the client node (the CI server running scripts on Jenkins on another CI system).

To offload an analysis, enter:

```
polyspaceserverroot\polyspace\bin\polyspace-bug-finder-server
  -batch -scheduler hostname|MJSName@hostname [options] [-mjs-username name]
```

where:

- *polyspaceserverroot* is the installation folder of Polyspace Server products on the client node, for instance, C:\Program Files\Polyspace Server\R2023a.
- *hostname* is the host name of the computer that hosts the head node of the MATLAB Parallel Server cluster.

MJSName is the name of the MATLAB Job Scheduler on the head node host.

If you use the `startjobmanager` command to start the MATLAB Job Scheduler, *MJSName* is the argument of the option `-name`.

- *options* are the Polyspace analysis options. These options are the same as that of a local analysis. For instance, you can use these options:
 - `-sources-list-file`: Specify a text file that has one source file name per line.
 - `-options-file`: Specify a text file that has one option per line.
 - `-results-dir`: Specify a download folder for storing results after analysis.

For the full list of options, see “Complete List of Polyspace Bug Finder Analysis Engine Options”.

- *name* is the user name required for job submissions using MATLAB Parallel Server. This credential is required only if you use a security level of 1 or higher for MATLAB Parallel Server submissions. See “Set MATLAB Job Scheduler Cluster Security” (MATLAB Parallel Server).

For security levels 2 and higher, you have to provide a password in a graphical window at the time of job submission. To avoid this prompt in the future, you can specify that the password be remembered on the computer.

The analysis executes locally on the CI server up to the end of the compilation phase. After compilation, the analysis job is submitted to the other server. On completion, the analysis results are downloaded back to the CI server. You can then upload the results to Polyspace Access for review, or report the results in some other format.

See Also

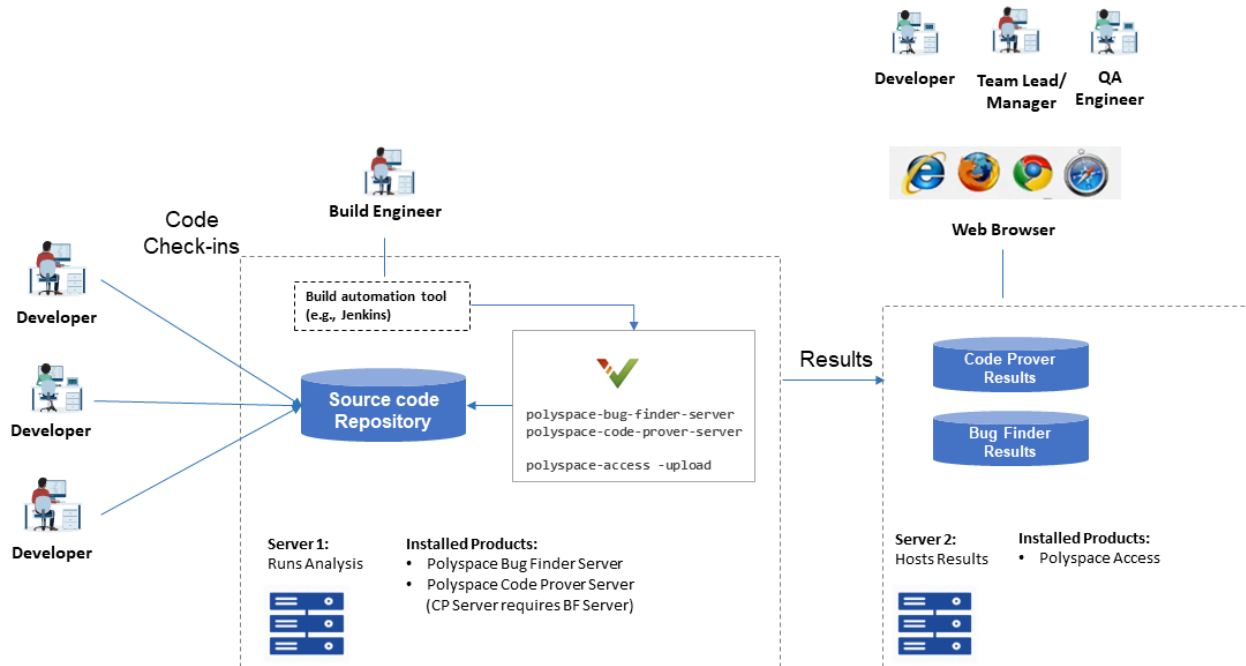
`polyspace-access`

More About

- “Run Polyspace Bug Finder on Server and Upload Results to Web Interface”

Sample Scripts for Polyspace Analysis with Jenkins

In a continuous integration process, developers submit code to a shared repository. An automated build system using a tool such as Jenkins builds and tests each submission at regular intervals or based on predefined triggers and integrates the code. You can run a Polyspace analysis as part of this process.



Note:

- Depending on the specifications, the same computer can serve as both Server 1 and Server 2.
- Though a server hosts the components for Polyspace web interface, each reviewer requires a PolyspaceAccess license to login to the interface.

This topic provides sample Shell scripts that run a Polyspace analysis using Polyspace Bug Finder Server and upload the results for review in the Polyspace Access web interface. The script also sends e-mail notifications to potential reviewers. Notified reviewers can login to the Polyspace Access web interface (if they have a Polyspace Access license) and review the results.

Extending Sample Scripts to Your Development Process

The scripts are written for a specific development toolchain but can be easily extended to the processes used in your project, team or organization. The scripts are also meant to be run in a Jenkins freestyle project. If you are using Jenkins Pipelines, see “Sample Jenkins Pipeline Scripts for Polyspace Analysis” on page 8-31.

In particular, the scripts:

- *Run on Linux only.*

The scripts use some Linux-specific commands such as `export`. However, these commands are not an integral part of the Polyspace workflow. If you write Windows scripts (.bat files), use the equivalent Windows commands instead.

- *Work only with Jenkins after you install the Polyspace plugin.*

The scripts are designed for the Jenkins plugin in these two ways:

- The scripts uses helper functions `$ps_helper` and `$ps_helper_access` for simpler scripting. The helper functions export Polyspace results for e-mail attachments and use command-line utilities to filter the results.

These helper functions are available only with the Jenkins plugin. However, the underlying commands come with a Polyspace Bug Finder Server installation. On build automation tools other than Jenkins, you can create these helper functions using the `polyspace-report-generator` command or `polyspace-access` command (with the `-export` option). See “Send Email Notifications with Polyspace Bug Finder Server Results”.

If you perform a distributed build in Jenkins, the plugin must be installed in the same folder in the same operating system on both the master node and the agent node executing the Polyspace analysis. Otherwise, you cannot use the helper functions.

- The scripts create text files for e-mail attachments and mail subjects and bodies for personalized e-mails. If you install the Polyspace plugin in Jenkins, an extension of an e-mail plugin is available for use in your Jenkins projects. The e-mail plugin allows you to easily send the personalized e-mails with the previously created subjects, bodies and attachments. Without the Polyspace plugin, you have to find an alternative way to send the e-mails.
- *Run a Bug Finder analysis.*

The scripts run Bug Finder on the demo example `Bug_Finder_Example`. If you install the product Polyspace Bug Finder Server, the folder containing the demo example is `polyspaceserverroot/polyspace/examples/cxx/Bug_Finder_Example`. Here, `polyspaceserverroot` is the installation folder for Polyspace Server products, for instance, `/usr/local/Polyspace Server/R2019a/`.

You can easily adapt the script to run Code Prover. Replace `polyspace-bug-finder-server` with `polyspace-code-prover-server`. You can use the demo example `Code_Prover_Example` specifically meant for Code Prover.

Prerequisites

To run a Polyspace analysis on a server and review the results in the Polyspace Access web interface, you must perform a one-time setup.

- To run the analysis, you must install one instance of the Polyspace Server product.
- To upload results, you must set up the components required to host the web interface of Polyspace Access.
- To view the uploaded results, you (and each developer reviewing the results) must have one Polyspace license.

Similar requirements apply to a Polyspace Code Prover analysis on a server.

See “Install Polyspace Server and Access Products”.

To install the Polyspace plugin, in the Jenkins interface, select **Manage Jenkins** on the left. Select **Manage Plugin**. Search for the Polyspace plugin and then download and install the plugin.

Set Up Polyspace Plugin in Jenkins

The following steps outline how to set up a Polyspace analysis in Jenkins after installing the Polyspace plugin. Note that the steps refer to Jenkins version 2.150.1. The steps in your Jenkins version and your Polyspace plugin installation might be slightly different.

If you use a different build automation tool, you can perform similar setup steps.

Specify Paths to Polyspace Commands and Server Details for Polyspace Access Web Interface

Specify the full paths of the folder containing the Polyspace commands and host name and port number of the server hosting the Polyspace Access web interface. After you specify the paths, in your scripts, you do not have to use the full paths to the commands or the server details for uploading results.

- 1 In the Jenkins interface, select **Manage Jenkins** on the left. Select **Configure System**.
- 2 In the **Polyspace** section, specify the following:
 - Paths to Polyspace commands.

The path refers to *polyspaceserverroot*/polyspace/bin, where *polyspaceserverroot* is the installation folder for Polyspace Server products, for instance, /usr/local/Polyspace Server/R2019a/.

The screenshot shows the 'Polyspace Bin' configuration form in Jenkins. It has a title 'Polyspace Bin' with a grid icon. There are two input fields: 'Name' with the value 'Server_install' and 'Binary Path' with the value '/usr/local/Polyspace Server/R2019a/polyspace/bin'. Below the 'Binary Path' field, the text 'Correct Configuration' is displayed. A red 'Delete' button is located at the bottom right of the form.

- The host name, port number and protocol (http or https) used by the server hosting the Polyspace Access web interface.

The screenshot shows the 'Polyspace Access' configuration form in Jenkins. It has a title 'Polyspace Access' with a grid icon. There are four input fields: 'Name' with the value 'Polyspace_Access', 'Protocol' with the value 'https', 'Host' with the value 'doc-server', and 'Port' with the value '9443'. A red 'Delete' button is located at the bottom right of the form.

The **Name** field allows you to define a convenient shorthand that you use later in Jenkins projects.

- 3 In the **E-mail Notification** section, specify your company's SMTP server (and other details needed for sending e-mails).

E-mail Notification	
SMTP server	<input type="text" value="mail.companyname.com"/>
Default user e-mail suffix	<input type="text"/>
<input type="checkbox"/> Use SMTP Authentication	
Use SSL	<input type="checkbox"/>
SMTP Port	<input type="text" value="25"/>
Reply-To Address	<input type="text"/>
Charset	<input type="text" value="UTF-8"/>
<input type="checkbox"/> Test configuration by sending test e-mail	

Create Jenkins Project for Running Polyspace

When you create a Jenkins project (for instance, a Freestyle project), you can refer to the Polyspace paths by the global shorthands that you defined earlier.

To create a Jenkins project for running Polyspace:

- 1 In the Jenkins interface, select **New Item** on the left. Select **Freestyle Project**.
- 2 In the **Build Environment** section of the project, enter the two shorthand names you defined earlier:
 - The name for the path to the folder containing the Polyspace commands
 - The name for the details of the server hosting the Polyspace Access web interface.

Also, enter a login and password that can be used to upload to the Polyspace Access web interface. The login and password must be associated with a Polyspace Access license.

Build Environment

Polyspace - Configuration to use

Polyspace Bin Configuration

Polyspace Access Configuration

Polyspace Access Credentials

- 3 In the **Build** section of the project, you can enter scripts that use the Polyspace commands and details of the server hosting the Polyspace Access web interface. The scripts run a Polyspace analysis and upload results to the Polyspace Access web interface.

Build

Execute shell

```
Command  set -e
         export RESULT=ResultBF
         export PROG=Bug_Finder_Example_2
         export PARENT_PROJECT=testProject
         rm -rf Notification && mkdir -p Notification

         build_cmd="gcc -c sources/*.c"
         polyspace-configure \
           -allow-overwrite \
           -allow-build-error \
           -prog $PROG \
           -author jenkins \
           -output-options-file $PROG.psopts \
           $build_cmd

         polyspace-bug-finder-server -options-file $PROG.psopts -results-dir $RESULT
```

- 4 In the **Post-build Actions** section of the project, configure e-mail addresses and attachments to be sent after the analysis.

Post-build Actions

Polyspace Notification X

Send to Recipients ?

Recipients

File to attach

Mail Subject

Mail Body

Script to Run Bug Finder, Upload Results and Send Common Notification

This script runs a Bug Finder analysis, uploads the results and exports defects with high impact for a common notification email to all recipients.

The script assumes that the current folder contains a folder `sources` with `.c` files. Otherwise modify the line `gcc -c sources/*.c` with the full path to the sources.

```

set -e
export RESULT=ResultBF
export PROG=Bug_Finder_Example
export PARENT_PROJECT=/public/BugFinderExample_PRS_01

# =====
# Trace build command and create an options file

build_cmd="gcc -c sources/*.c"
polyspace-configure \
    -allow-overwrite \
    -allow-build-error \
    -prog $PROG \
    -author jenkins \
    -output-options-file $PROG.psopts \
    $build_cmd

# =====
# Run Bug Finder on the options file

polyspace-bug-finder-server -options-file $PROG.psopts -results-dir $RESULT

# =====
# Upload results to Polyspace Access web interface

$ps_helper_access -create-project $PARENT_PROJECT
$ps_helper_access \
    -upload $RESULT \
    -parent-project $PARENT_PROJECT \
    -project $PROG

# =====
# Export results filtered for defects with "High" impact

$ps_helper_access \
    -export $PARENT_PROJECT/$PROG \
    -output Results_All.tsv \
    -defects High

# =====
# Finalize Jenkins status

exit 0

```

After the script is run, you can create a post-build action to send an e-mail to all recipients with the exported file `Results_All.tsv`.

Post-build Actions

Polyspace Notification
X

Send to Recipients ?

Recipients

File to attach

Mail Subject

Mail Body

In this script, `$ps_helper_access` is a shorthand for the `polyspace-access` command with the options specifying host name, port, login and encrypted password included. The other `polyspace-access` options are explicitly written in the script.

Script to Run Bug Finder, Upload Results and Send Personalized Notification

This script runs the previous Bug Finder analysis and uploads the results. However, the script differs from the previous script in these ways:

- The script uses a `run_command` function that prints a message when running a command. The function helps determine from the console output which part of the script is running.
- When exporting the results, the script creates a separate results file for different owners.
 - A main file `Results_All.tsv` contains all results. This file is sent in e-mail attachment to a manager. The manager email is configured in the post-build step.

If the file contains more than 10 defects, the build status is considered as a failure. The script sends a status `UNSTABLE` in the e-mail notification.

- The results file `Results_Users_userA.tsv` exported for `userA` contains defects from the group Programming and with impact High.

This result file is sent in e-mail attachment to `userA`.

- The results file `Results_Users_userB.tsv` exported for `userB` contains defects from the function `bug_memstdlib()`.

This result file is sent in e-mail attachment to `userB`.

- A separate mail subject is created for the manager in the file `mailsubject_manager.txt` and for users `userA` and `userB` in the files `mailsubject_user_userA.txt` and `mailsubject_user_userB.txt` respectively.

A mail body is created for the email to the manager in the file `mailbody_manager.txt`.

The script:

- Assumes that the current folder contains a folder `sources` with `.c` files.

Otherwise, modify the line `gcc -c sources/*.c` with the full path to the sources.

- Assumes users named `userA` and `userB`. In particular, the email addresses `userA@companyname.com` and `userB@companyname.com` (determined from the user name and SMTP server configured earlier) must be real e-mail addresses.

Replace the names with real user names.

```

set -e
export RESULT=ResultBF
export PROG=Bug_Finder_Example
export REPORT=Results_List.tsv

# =====
# Define function to print message while running command
run_command()
{
# $1 is a message
# $2 $3 ... is the command to dump and to run
message=$1
shift
cat >> mailbody_manager.txt << EOF
$(date): $message

EOF
"$@"
}

# =====
# Initialize mail body
cat > mailbody_manager.txt << EOF
Dear Manager(s)

Here is the report of the Jenkins Job ${JOB_NAME} #${BUILD_NUMBER}
It contains all Red Defect found in Bug Finder Example project

EOF

# =====
# Trace build command and create options file

build_cmd="gcc -c sources/*.c"
run_command "Tracing build command", \
            polyspace-configure \
            -allow-overwrite \
            -allow-build-error \
            -prog $PROG \
            -author jenkins \
            -output-options-file $PROG.psopts \
            $build_cmd

# =====
# Run Bug Finder on the options file

run_command "Running Bug finder" \
            polyspace-bug-finder-server -options-file $PROG.psopts \
            -results-dir $RESULT

# =====
# Upload results to Polyspace Access web interface

run_command "Creating Project $PARENT_PROJECT" \

```

```

$ps_helper_access -create-project $PARENT_PROJECT

run_command "Uploading on $PARENT_PROJECT/$PROG" \
  $ps_helper_access \
    -upload $RESULT \
    -parent-project $PARENT_PROJECT \
    -project $PROG \
    -output upload.output
PROJECT_RUNID=$(($ps_helper prs_print_runid upload.output)
PROJECT_URL=$(($ps_helper prs_print_projecturl upload.output $POLYSPACE_ACCESS_URL)

# =====
# Export report

run_command "Exporting report from $PARENT_PROJECT/$PROG" \
  $ps_helper_access \
    -export $PROJECT_RUNID \
    -output $REPORT \
    -defects High

# =====
# Filter Reports

run_command "Filtering reports for defects" \
  $ps_helper report_filter \
    $REPORT \
    Results_All.tsv \
    Family Defect \

# =====
# Filter Reports for userA and userB

run_command "Filtering Reports for userA based on Group and Information" \
  $ps_helper report_filter \
    $REPORT \
    Results_Users.tsv \
    userA \
    Group Programming \
    Information "Impact: High"
run_command "Filtering Reports for userB based on Function" \
  $ps_helper report_filter \
    $REPORT \
    Results_Users.tsv \
    userB \
    Function "bug_memstdlib()"

# =====
# Update Jenkins status
# Jenkins build status is unstable when there are more than 10 Defects

BUILD_STATUS=$(($ps_helper report_status Results_All.tsv 10)

# =====
# Update mail body and mail subject

```

```

NB_FINDINGS_ALL=$(($ps_helper report_count_findings Results_All.tsv)
NB_FINDINGS_USERA=$(($ps_helper report_count_findings Results_Users_userA.tsv)
NB_FINDINGS_USERB=$(($ps_helper report_count_findings Results_Users_userB.tsv)
cat >> mailbody_manager.txt << EOF

Number of defects: $NB_FINDINGS_ALL
Number of findings owned by userA: $NB_FINDINGS_USERA
Number of findings owned by userB: $NB_FINDINGS_USERB

All results are uploaded in: $PROJECT_URL

Build Status: $BUILD_STATUS

EOF

cat >> mailsubject_manager.txt << EOF
Polyspace run completed with status $BUILD_STATUS and $NB_FINDINGS_ALL findings
EOF

for user in userA userB
do
echo "$user - $($ps_helper report_count_findings Results_Users_$user.tsv) findings"\
  > mailsubject_user_$user.txt
done

# =====
# Exit with correct build status

[ "$BUILD_STATUS" != "SUCCESS" ] && exit 129
exit 0

```

After the script is run, you can create a post-build action to send an e-mail to a manager with the exported file `Results_All.tsv`. Specify the e-mail address in the **Recipients** field, the email subject in the **Mail Subject** field and the email body in the **Mail Body** field.

In addition, a separate e-mail is sent to userA and userB with the files `Results_Users_userA.tsv` and `Results_Users_userB.tsv` in attachment (and the content of `mailsubject_user_userA.txt` and `mailsubject_user_userB.txt` as mail subjects). The e-mail addresses are `userA@companyname.com` and `userB@companyname.com` (determined from the user name and SMTP server configured earlier).

Post-build Actions

Polyspace Notification
X

Send to Recipients ?

Recipients

File to attach

Mail Subject

Mail Body

Send to Owners ?

Query Base Name

Mail Subject Base Name

Mail Body Base Name

Unique recipients - Debug only

Add post-build action ▼

The script uses the helper function `$ps_helper` to filter the results based on group, impact and function. The helper function uses command-line utilities to filter the main file for results and perform actions such as create a separate results file for each owner. The function takes these actions as arguments:

- `report_filter`: Filters results from exported text file based on contents of the text file.

For instance:

```
$ps_helper report_filter \
    Results_List.tsv \
    Results_Users.tsv \
    userA \
    Group Programming \
    Information "Impact: High"
```

reads the file `Results_List.tsv` and writes to the file `Results_Users_userA.tsv`. The text file `Results_List.tsv` contains columns for Group and Information. Only those rows where the Group column contains Programming and the Information column contains Impact: High are written to the file `Results_Users_userA.tsv`.

- `report_status`: Returns UNSTABLE or SUCCESS based on the number of results in a file.

For instance:

```
BUILD_STATUS=$(($ps_helper report_status Results_All.tsv 10))
```

returns UNSTABLE if the file `Results_All.tsv` contains more than 10 results (10 rows).

- `report_count_findings`: Reports number of results in a file.

For instance:

```
NB_FINDINGS_ALL=$(($ps_helper report_count_findings Results_All.tsv)
```

returns the number of results (rows) in the file `Results_All.tsv`.

- `prs_print_projecturl`: Uses the host name and port number to create the URL of the Polyspace Access web interface.

For instance:

```
PROJECT_URL=$(($ps_helper prs_print_projecturl Results_All.tsv $POLYSPACE_ACCESS_URL)
```

reads the file `Results_All.tsv` (exported by the `polyspace-access` command) and extracts the URL of the Polyspace Access web interface in `$POLYSPACE_ACCESS_URL` and the URL of the current project in `$PROJECT_URL`.

See Also

`polyspace-bug-finder-server` | `polyspace-code-prover-server` | `polyspace-report-generator` | `polyspace-access` | `polyspace-configure`

More About

- “Run Polyspace Bug Finder on Server and Upload Results to Web Interface”
- “Send Email Notifications with Polyspace Bug Finder Server Results”
- “Sample Jenkins Pipeline Scripts for Polyspace Analysis” on page 8-31
- “Offload Polyspace Analysis from Continuous Integration Server to Another Server” on page 8-12

Sample Jenkins Pipeline Scripts for Polyspace Analysis

Jenkins Pipelines enable automating the workflow of a continuous delivery pipeline through scripts in Jenkins. You can write Pipeline scripts that build projects, run test suites and perform all necessary checks before your code is ready for shipping. You can check in these scripts as part of a version control system and subject them to the same review and versioning as the code itself.

You can run a Polyspace analysis in a Jenkins Pipeline script. If you are using Freestyle Projects instead of Pipelines in Jenkins, use the Polyspace plugin for scripting conveniences. See “Sample Scripts for Polyspace Analysis with Jenkins” on page 8-17. If you are using Pipelines, modify the script provided below to run a Polyspace analysis.

Prerequisites

To run a Polyspace analysis on a server and review the results in the Polyspace Access web interface, you must perform a one-time setup.

- To run the analysis, you must install one instance of the Polyspace Server product.
- To upload results, you must set up the components required to host the web interface of Polyspace Access.
- To view the uploaded results, you and each developer reviewing the results must have one Polyspace license.

See “Install Polyspace Server and Access Products”.

Run Polyspace Analysis in Stages in a Pipeline Script

To create a Jenkins Pipeline script:

- 1 In the Jenkins interface, select **New Item** on the left. Select **Pipeline**.
- 2 In the **Pipeline** section of the project, select Pipeline script for **Definition**. Enter this script.

The parts in bold indicate places where you have to modify the script for your source code and Polyspace installation.

The script is not available in the PDF documentation. Search for **Polyspace Jenkins Pipelines** in the MathWorks online documentation and copy the script from the online version of this page.

When you build this project, you can see the various stages of the analysis like this:

Prepare	Checkout	Configure	Analyze	Upload	Notification
1s	1s	14s	4min 22s	1min 32s	369ms
1s	1s	14s	4min 22s	1min 32s	369ms

This script can be part of a larger script that you save in a Jenkinsfile and commit to your version control system. See Using a Jenkinsfile.

You can modify the script as needed:

- The script runs each step of the Polyspace analysis workflow in a separate `stage` section. You can combine several steps together in one `stage`.
- The script runs Linux Shell commands by using the `sh` directive. You can run Windows commands by using the `bat` directive instead.
- The script uses data from the Credentials plugin to extract user name and password. If you save credentials in some other form, you can replace the `withCredentials` command that binds user credentials to variables.
- The script builds source code using a makefile on a Git sandbox with this `make` command:

```
make -C $git_sandbox
```

If you use a different build command, you can replace this line with your build command.

For more information on the Pipeline-specific syntax in this script, see:

- Pipeline Syntax: Describes `node`, `stage`, `label`.
- Pipeline Steps Reference: Describes `sh`, `mail`.
- Credentials Binding Plugin: Describes `withCredentials`.

For more information on the Polyspace commands in this script, see:

- `polyspace-configure`
- `polyspace-bug-finder-server` (also `polyspace-code-prover-server`)
- `polyspace-access`

See Also

“Sample Scripts for Polyspace Analysis with Jenkins” on page 8-17

Integrate Polyspace Server Products with MATLAB

You can install Polyspace Bug Finder Server and Polyspace Code Prover Server as standalone products and analyze C/C++ code.

When installing Polyspace server products and MATLAB, you cannot install MATLAB and Polyspace server products together in a single run of the installer. First install MATLAB by running the MATLAB installer. Then install the Polyspace server product in a different root folder by running the installer separately. For instance, in Windows:

- Your default MATLAB root folder is C:\Program Files\MATLAB\R2023a.
- Your default Polyspace root folder is C:\Program Files\Polyspace Server\R2023a for the Polyspace server products.

To automate the Polyspace analysis by using MATLAB scripts, integrate the Polyspace server products and MATLAB by running a post-installation step.

Integrate Polyspace Server Products with MATLAB

You can integrate your Polyspace server product with MATLAB only if both installations are from the same release. After the integration, you can use all MATLAB functions and classes available for running Polyspace.

To link your MATLAB and Polyspace installations:

- 1 Open MATLAB with administrator privileges.
- 2 At the MATLAB command prompt, enter:

```
polyspacesetup('install');
```

By default, Polyspace is installed in the folder C:\Program Files\Polyspace\R2023a. If you install Polyspace in the default folder, the command integrates Polyspace with MATLAB. If a Polyspace installation is not detected at the default location, provide the path to the Polyspace installation folder when prompted. The process might take a few minutes to complete.

To avoid the prompt during installation, enter:

```
polyspacesetup('install','polyspaceFolder', Folder, 'silent', true);
```

- 3 Restart MATLAB. You can now use all functions and classes available for running Polyspace server products.

A MATLAB installation can be integrated with only one Polyspace installation. To integrate to a new Polyspace installation, any previous integration must be removed. To remove the integration between a Polyspace and MATLAB installation, open MATLAB with administrator privilege and at the MATLAB command prompt, enter:

```
polyspacesetup('uninstall')
```

Check Integration Between MATLAB and Polyspace

To check if a MATLAB installation is already integrated with a Polyspace installation, open MATLAB and at the command prompt, enter:

```
ver
```

You see the list of products installed. If Polyspace is integrated with MATLAB, you can see the Polyspace products in the list.

The MATLAB-Polyspace integration adds some Polyspace installation subfolders to the MATLAB search path. To see which paths were added, enter:

```
polyspacesetup('showpolyspacefolders')
```

Run Polyspace Server Products with MATLAB Scripts

In a continuous integration process, you can execute MATLAB scripts that run a Polyspace analysis on new code submissions and compares the results against predefined criteria. Use these functions/classes:

- Create a `polyspace.Project` object to configure Polyspace analysis options, run an analysis and read results to MATLAB tables. You can use other MATLAB functions for comparing results against predefined criteria.

To only read existing results without running an analysis, use the `polyspace.BugFinderResults` class with the path to a results folder.

- If you want a more granular selection of checkers for:
 - Coding rules, create a `polyspace.CodingRulesOptions` object.
 - Bug Finder defects, create a `polyspace.DefectsOptions` object.

To create a custom target for the analysis and explicitly specify sizes of data types, create a `polyspace.GenericTargetOptions` object.

You can also use the `polyspaceBugFinderServer` function to run the analysis and then read results with the `polyspace.BugFinderResults` class. If you use build commands to build your source code, you can create a Polyspace configuration from the build command using the `polyspaceConfigure` function.

See Also

`polyspacesetup`

Configure Job Submissions from Desktop to Server

Offload Polyspace Analysis to Remote Servers from Desktop

- “Send Polyspace Analysis from Desktop to Remote Servers” on page 9-2
- “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts” on page 9-5

Send Polyspace Analysis from Desktop to Remote Servers

In this section...

“Client-Server Workflow for Running Analysis” on page 9-2

“Prerequisites” on page 9-3

“Offload Analysis in Polyspace User Interface” on page 9-3
--

You can perform a Polyspace analysis locally on your desktop or offload the analysis to one or more dedicated servers. You offload a Polyspace analysis from a Polyspace desktop product such as Polyspace Bug Finder but the analysis runs on the server using a Polyspace server product such as Polyspace Bug Finder Server.

This topic shows how to send a Polyspace analysis from the user interface of the Polyspace desktop products.

- To offload an analysis with scripts, see “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts” on page 9-5.
- For a simple tutorial that walks through all the steps for offloading a Polyspace analysis, see “Send Bug Finder Analysis from Desktop to Locally Hosted Server”. In the tutorial, the same computer acts as the client and the server.

Client-Server Workflow for Running Analysis

After the initial setup, you can submit a Polyspace analysis from a client desktop to a server. The client-server workflow happens in three steps. All three steps can be performed on the same computer or three different computers.

- 1 Client node:** You specify Polyspace analysis options and start the analysis on the client desktop. The initial phase of analysis up to compilation runs on the desktop. After compilation, the analysis job is submitted to the server.

You require the Polyspace desktop product, Polyspace Bug Finder on the computer that acts as the client node.

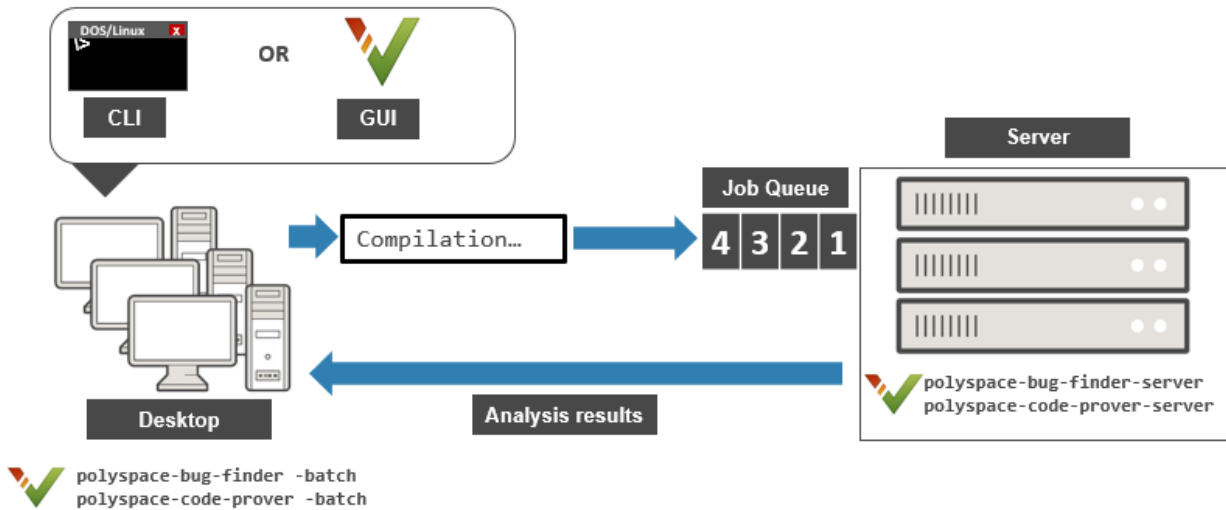
- 2 Head node:** The server consists of a head node and several worker nodes. The head node uses a job scheduler to manage submissions from multiple client desktops. The jobs are then distributed to the worker nodes as they become available.

You require the product MATLAB Parallel Server on the computer that acts as the head node.

- 3 Worker nodes:** When a worker becomes available, the job scheduler assigns the analysis to the worker. The Polyspace analysis runs on the worker and the results are downloaded back to the client desktop for review.

You require the product MATLAB Parallel Server on the computers that act as worker nodes. You also require the Polyspace server products, Polyspace Bug Finder Server and/or Polyspace Code Prover Server, to run the analysis.

Note The versions of Polyspace on the client and worker nodes must match.



Prerequisites

Before offloading an analysis from the user interface of the Polyspace desktop products, you must set up your project's source files, analysis options, and remote analysis settings. If you have not done so, for more information on:

- How to add source files, see “Add Source Files for Analysis in Polyspace Desktop User Interface” on page 2-2.
- How to set up communication between client and server, see “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

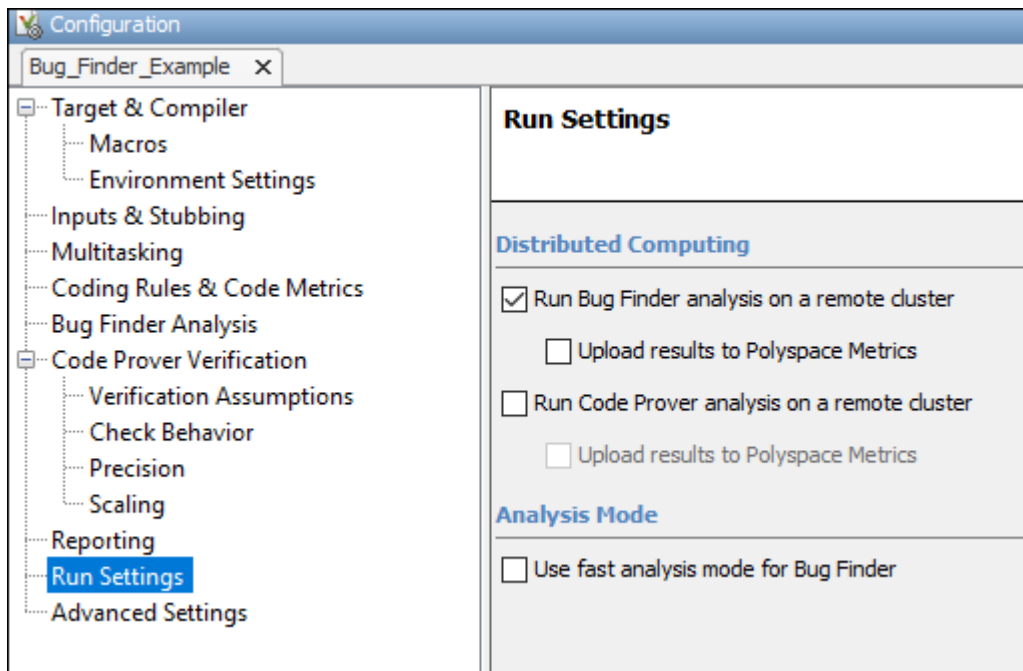
Once you have set up a Polyspace project and established communication between a desktop and a remote server, you are ready to offload a Polyspace analysis.

Offload Analysis in Polyspace User Interface

To start a remote analysis:

- 1 Select a project to analyze.
- 2 On the **Configuration** pane, select **Run Settings**.

Select **Run Bug Finder analysis on a remote cluster** and/or **Run Code Prover analysis on a remote cluster**.



- 3 Start the analysis. For instance, to start a Bug Finder analysis, click the **Run Bug Finder** button.

The compilation part of the analysis takes place on the desktop product. After compilation, the analysis is offloaded to the server.

- 4 To monitor the analysis, select **Tools > Open Job Monitor**. In the Polyspace Job Monitor, follow your queued job to monitor progress.

Once the analysis is complete, the results are downloaded back to the user interface of the Polyspace desktop products. You can open the results directly in the user interface.

If the analysis stops after compilation and you have to restart the analysis, to avoid restarting from the compilation phase, use the option `-submit-job-from-previous-compilation-results`.

See Also

Run Bug Finder or Code Prover analysis on a remote cluster (-batch)

More About

- “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”
- “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts” on page 9-5

Send Polyspace Analysis from Desktop to Remote Servers Using Scripts

Instead of running a Polyspace analysis on your local desktop, you can send the analysis to a remote cluster. You can use a dedicated cluster for running Polyspace to free up memory on your local desktop.

This topic shows how to use Windows or Linux scripts to send the analysis to a remote cluster and download the results to your desktop after analysis.

- To offload an analysis from the Polyspace user interface, see “Send Polyspace Analysis from Desktop to Remote Servers” on page 9-2.
- For a simple tutorial that walks through all the steps for offloading a Polyspace analysis, see “Send Bug Finder Analysis from Desktop to Locally Hosted Server”. In the tutorial, the same computer acts as the client and the server.

Client-Server Workflow for Running Analysis

After the initial setup, you can submit a Polyspace analysis from a client desktop to a server. The client-server workflow happens in three steps. All three steps can be performed on the same computer or three different computers.

- 1 Client node:** You specify Polyspace analysis options and start the analysis on the client desktop. The initial phase of analysis up to compilation runs on the desktop. After compilation, the analysis job is submitted to the server.

You require the Polyspace desktop product, Polyspace Bug Finder on the computer that acts as the client node.

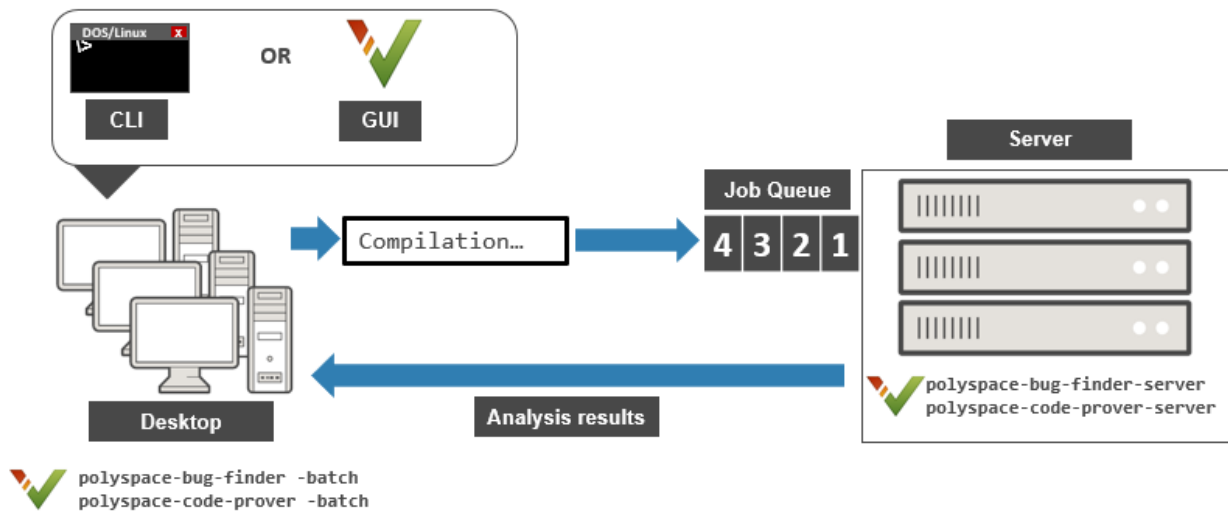
- 2 Head node:** The server consists of a head node and several worker nodes. The head node uses a job scheduler to manage submissions from multiple client desktops. The jobs are then distributed to the worker nodes as they become available.

You require the product MATLAB Parallel Server on the computer that acts as the head node.

- 3 Worker nodes:** When a worker becomes available, the job scheduler assigns the analysis to the worker. The Polyspace analysis runs on the worker and the results are downloaded back to the client desktop for review.

You require the product MATLAB Parallel Server on the computers that act as worker nodes. You also require the Polyspace server products, Polyspace Bug Finder Server and/or Polyspace Code Prover Server to run the analysis.

Note The versions of Polyspace on the client and worker nodes must match.



Prerequisites

Before you run a remote analysis by using scripts, you must set up communication between a desktop and a remote server. See “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

Run Remote Analysis

To run a remote analysis, use the following command. Here, [] indicates optional flags.

```
polyspaceroot\polyspace\bin\polyspace-bug-finder
-batch -scheduler NodeHost|MJSName@NodeHost [-wait -download]
[options] [-mjs-username name]
```

where:

- *polyspaceroot* is the installation folder of Polyspace desktop products, for instance, C:\Program Files\Polyspace\R2023a.
- *NodeHost* is the name of the computer that hosts the head node of the MATLAB Parallel Server cluster.

MJSName is the name of the MATLAB Job Scheduler on the head node host.

If you set up communications with a cluster from the Polyspace user interface, you can determine *NodeHost* and *MJSName* from the user interface.

Select **Tools > Preferences**, and then click **Settings** on the **Server Configuration** tab to open the **Cluster Profile Manager**. Select the cluster profile in the left pane, and see the **MJSName** and **Host** fields on the **Properties** tab for *MJSName* and *NodeHost*.

If you use the `startjobmanager` command to start the MATLAB Job Scheduler, *MJSName* is the argument of the option `-name`. For details, see “Configure Advanced Options for MATLAB Job Scheduler Integration” (MATLAB Parallel Server).

- *options* are the analysis options. These options are the same as that of a local analysis. For instance, you can use these options:
 - `-sources-list-file`: Specify a text file with one source file name per line.
 - `-options-file`: Specify a text file with one option per line.
 - `-results-dir`: Specify a download folder for storing results after analysis.

For the full list of options, see “Complete List of Polyspace Bug Finder Analysis Engine Options”. Alternatively, you can:

- Start an analysis in the user interface and stop after compilation. You can obtain the text files and scripts for running the analysis at the command line. See “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 4-15.
- Enter `polyspace-bug-finder -h`. The list of available options with a brief description are displayed.
- Place your cursor over each option on the **Configuration** pane in the Polyspace user interface. Click the **More Help** button for information on the option syntax and when the option is required.
- *name* is the username required for job submissions using MATLAB Parallel Server. These credentials are required only if you use a security level of 1 or higher for MATLAB Parallel Server submissions. See “Set MATLAB Job Scheduler Cluster Security” (MATLAB Parallel Server).

For security levels 2 and higher, you have to provide a password in a graphical window at the time of job submission. To avoid this prompt in the future, you can specify that the password be remembered on the computer.

The analysis happens in two parts:

- 1 The first part of the analysis up to the end of the compilation phase executes locally on your desktop. After compilation, the software submits the analysis job to the cluster and provides a job ID. You can also read the ID from the file `ID.txt`, which is stored in the `.status` subfolder of the results folder. To monitor your analysis, use the `polyspace-jobs-manager` command with the job ID.
- 2 The remaining part of the analysis continues on the cluster. The command waits till the analysis is completed and the results automatically downloaded back to the desktop. If you want to free up the console and download results later using the `polyspace-jobs-manager` command, omit the options `-wait -download`.

If the analysis stops after compilation and you have to restart the analysis, to avoid rerunning the compilation phase, use the option `-submit-job-from-previous-compilation-results`.

Manage Remote Analysis

To manage multiple remote analyses, use the option `-batch`. For instance:

```
polyspaceroot\polyspace\bin\polyspace-jobs-manager action
-scheduler schedulerName
```

See also Run Bug Finder or Code Prover analysis on a remote cluster (-batch). Here:

- *polyspaceroot* is your MATLAB installation folder.
- *schedulerName* is one of the following:
 - Name of the computer that hosts the head node of your MATLAB Parallel Server cluster (*NodeHost*).
 - Name of the MATLAB Job Scheduler on the head node host (*MJSName@NodeHost*).
 - Name of a MATLAB cluster profile (*ClusterProfile*).

For more information about clusters, see “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox)

If you do not specify a job scheduler, `polyspace-job-manager` uses the scheduler specified in the Polyspace preferences. To see the scheduler name, select **Tools > Preferences**. On the **Server Configuration** tab, see the **Job scheduler host name**.

- *action* refers to the possible action commands to manage jobs on the scheduler:
 - `listjobs`:

Generate a list of Polyspace jobs on the scheduler. For each job, the software produces this information:

- ID — Verification or analysis identifier.
- AUTHOR — Name of user that submitted job.
- APPLICATION — Name of Polyspace product, for example, Polyspace Code Prover or Polyspace Bug Finder.
- LOCAL_RESULTS_DIR — Results folder on local computer, specified through the **Tools > Preferences > Server Configuration** tab.
- WORKER — Local computer from which job was submitted.
- STATUS — Status of job, for example, running and completed.
- DATE — Date on which job was submitted.
- LANG — Language of submitted source code.
- `download -job ID -results-folder FolderPath`:

Download results of analysis with specified ID to folder specified by *FolderPath*. If you use the option `-wait -download` when sending the analysis job to a server, the results are automatically downloaded after analysis. Only when you want to explicitly download results do you need to use the `polyspace-jobs-manager` command with the `download` action.

When the analysis job is queued on the server, the command `polyspace-bug-finder` returns a job id. In addition, a file `ID.txt` that is stored in the `.status` subfolder of the results folder contains the job ID in this format:

```
job_id;server_name:project_name version_number
```

For instance, `92;localhost:Demo 1.0`.

If you do not use the `-results-folder` option, the software downloads the result to the folder that you specified when starting analysis, using the `-results-dir` option.

After downloading results, use the Polyspace user interface to view the results.

- `getlog -job ID:`

Open log for job with specified ID.

- `remove -job ID:`

Remove job with specified ID.

- `promote -job ID:`

Promote job with specified ID in the queue.

- `demote -job ID`

Demote job with specified ID in the queue.

Sample Scripts for Remote Analysis

In Windows, to avoid typing the commands each time, you can save the commands in a batch file. In Linux, you can relaunch the analysis by using a shell script. To create a batch file for running analysis:

- 1 Save your analysis options in a file `listoptions.txt`. See `-options-file`.
- 2 Create a file `launcher.bat` in a text editor like Notepad.

In the file, enter these commands:

```
echo off
set POLYSPACE_PATH=polyspaceroot\polyspace\bin
set RESULTS_PATH=C:\Results
set OPTIONS_FILE=C:\Options\listoptions.txt
"%POLYSPACE_PATH%\polyspace-bug-finder.exe" -batch -scheduler hostname
    -results-dir "%RESULTS_PATH%" -options-file "%OPTIONS_FILE%"
pause
```

`polyspaceroot` is the Polyspace installation folder. `hostname` is the name of the computer that hosts the head node of your MATLAB Parallel Server cluster.

- 3 Replace the definitions of these variables in the file:
 - `POLYSPACE_PATH`: Enter the actual location of the `.exe` file.
 - `RESULTS_PATH`: Enter the path to a folder. The files generated during compilation are saved in the folder.
 - `OPTIONS_FILE`: Enter the path to the file `listoptions.txt`.
- 4 Double-click `launcher.bat` to run the analysis.

Tip If you run a Polyspace analysis, a Windows `.bat` or Linux `.sh` file is generated. The file is in the `.settings` subfolder in your results folder. Instead of writing a script from scratch, you can relaunch the analysis using this file.

See Also

Run Bug Finder or Code Prover analysis on a remote cluster (`-batch`)

More About

- “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”
- “Send Polyspace Analysis from Desktop to Remote Servers” on page 9-2

Configure Analysis in IDEs

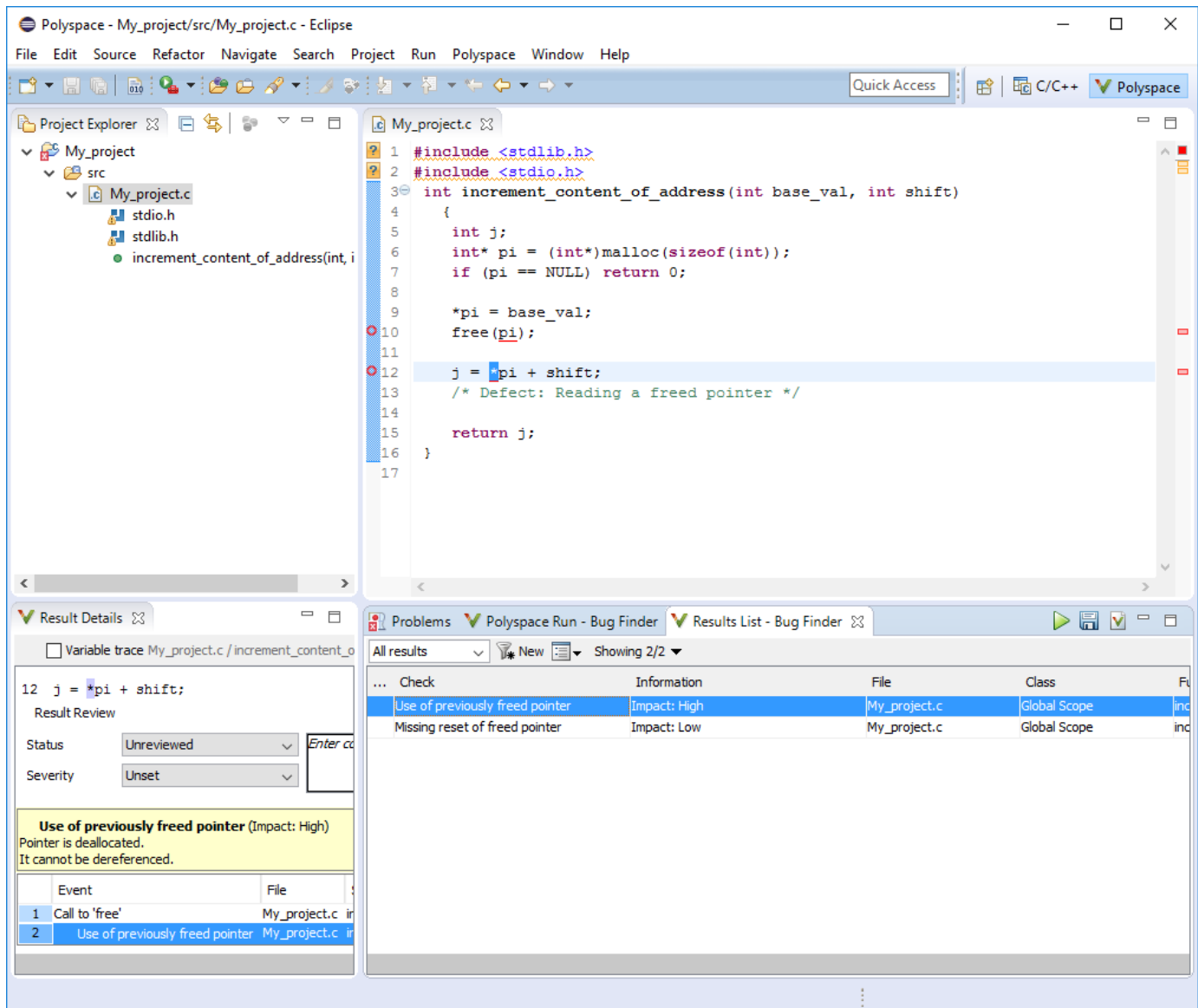
Run Polyspace Analysis in IDE Plugins

Run Polyspace Analysis on Eclipse Projects

This topic describes how to run a Polyspace analysis on complete Eclipse projects using Polyspace Bug Finder or Polyspace Code Prover. For the Polyspace as You Code plugin, see “Run Polyspace as You Code in Eclipse and Review Results”.

If you develop code in Eclipse or an Eclipse-based IDE, you can install the Polyspace plugin and run a Polyspace analysis on the source files in an Eclipse project. You can check for bugs each time you save your code, or explicitly run an analysis.

This topic describes how to set up a Polyspace analysis in Eclipse and review analysis results.



After you install the Polyspace plugin, you see a **Polyspace** menu and right-click options in the **Project Explorer** to run a Polyspace analysis.

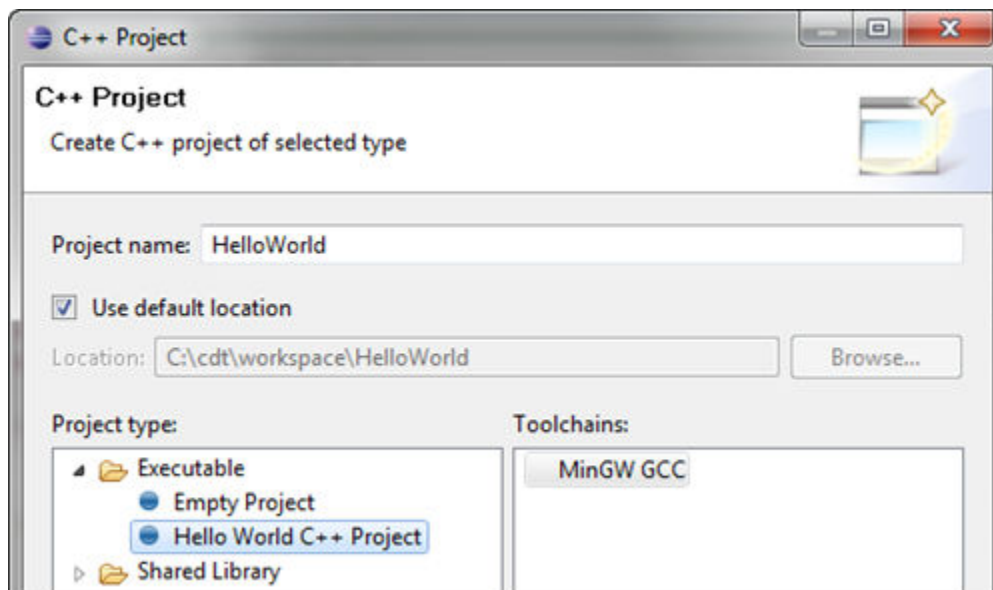
The analysis progress bar, quick run buttons and analysis results appear on specific panes. To avoid cluttering your window, you can confine these panes to the Polyspace perspective. Select **Window > Open Perspective > Other**. In the Open Perspective dialog box, select **Polyspace**. You can switch back to other perspectives using tabs on the upper right.

Configure and Run Analysis

Configure analysis

Polyspace analyzes the source files in your Eclipse project. In addition to sources, the analysis uses the following information:

- **Compiler:** The compiler toolchain can be extracted from your Eclipse project. If the project directly refers to a compilation toolchain such as MinGW GCC, the Polyspace analysis can use the information.

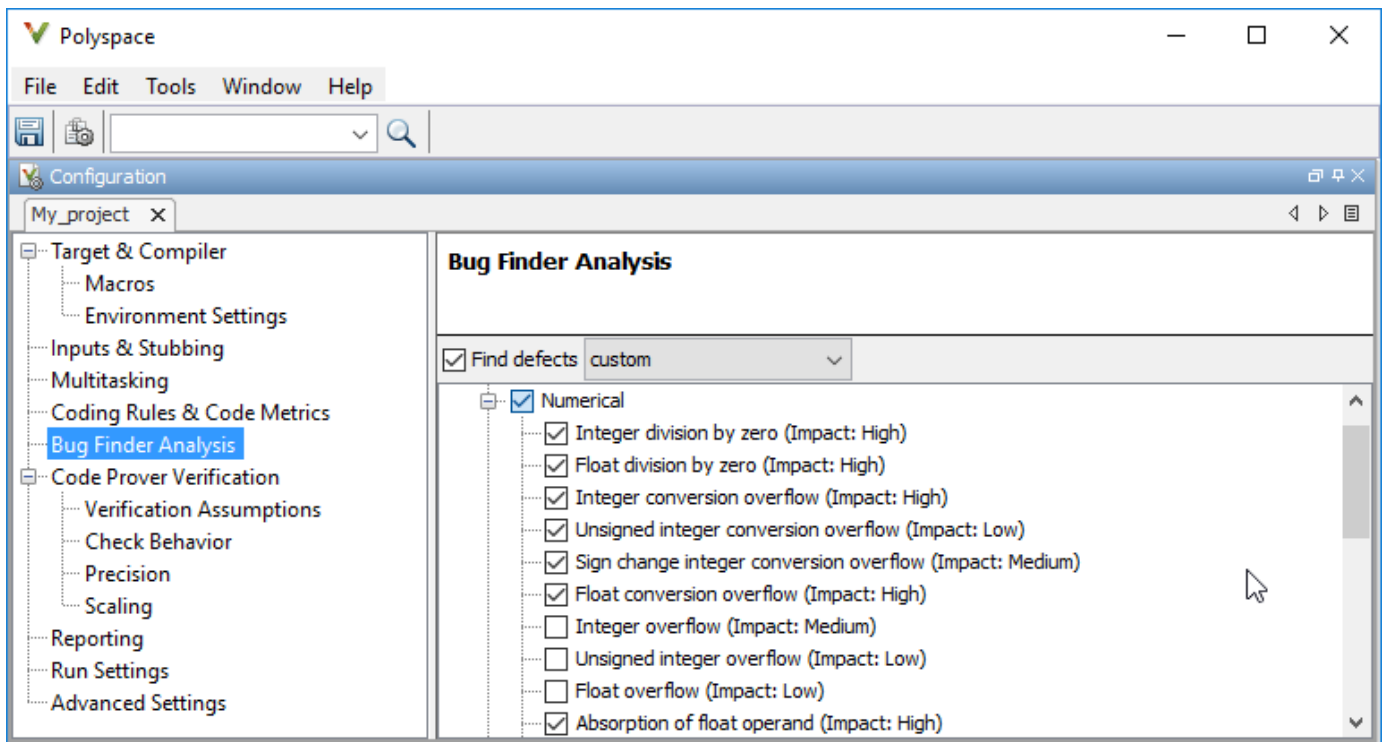


If your Eclipse project uses a build command (makefile) that has the compiler information, you must perform some additional steps to extract this information for the Polyspace analysis.

If Polyspace cannot extract the compiler information from your build command, you can also explicitly specify your compiler options explicitly like other analysis options.

See “Specify Polyspace Compiler Options Through Eclipse Project” on page 10-7.

- **Other analysis options:** You can retain the default analysis options or adjust them to your requirements. Select **Polyspace > Configure Project**.




The key options are:

- **Target & Compiler:** If you have not specified your compiler information through your Eclipse project, use these options.
- **Bug Finder Analysis:** Specify which defects to check for in a Bug Finder analysis.
- **Code Prover Verification, Check Behavior, Precision:** Modify the behavior of checkers in a Code Prover verification.

Note that you cannot run a remote analysis using the Polyspace plugin for Eclipse. To send the analysis job to a remote cluster, start the analysis from the Polyspace user interface or using scripts. See “Bug Finder Analysis on Clusters”.

Run analysis

After configuration, you can start and stop a Polyspace analysis explicitly from the **Polyspace** menu, right-click options on your Eclipse project or quick run buttons in the Polyspace panes. You can switch between Bug Finder and Code Prover using the  icon on the **Polyspace Run** pane.

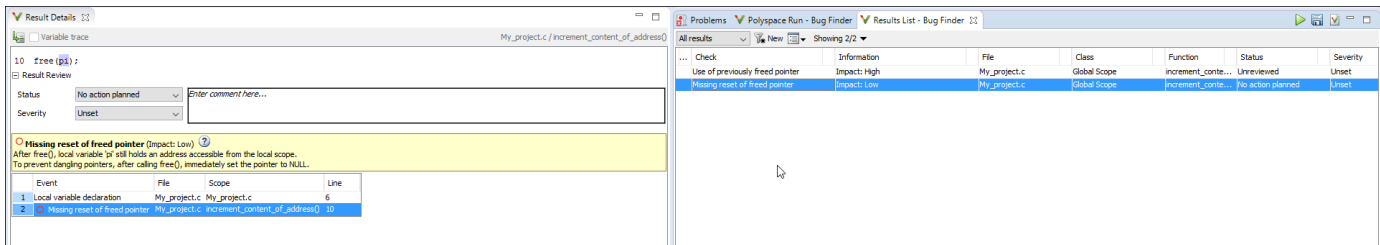
Run analysis when saving code

In the Polyspace perspective, you can set up a Bug Finder analysis that runs each time you save your code. To enable this analysis, select **Polyspace > Run Fast Analysis on Save**. The analysis runs quickly but looks for a reduced set of defects. You get the same results as if you had specified the analysis option Use fast analysis mode for Bug Finder (`-fast-analysis`).



Review Analysis Results

View results after analysis

After analysis, the results appear on the **Results List** pane. Click each result to see the source code and details on the **Result Details** pane.



View results as available

Some results of a Bug Finder analysis are often available before the analysis is complete. If so, the  icon in the **Polyspace Run - Bug Finder** pane turns to . To load available results, click this icon. The icon shows up again when more results are available.

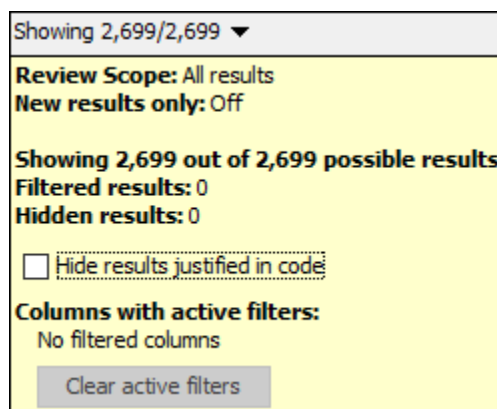
Address results

Based on the result details, fix your code or justify the result. To justify a result, set its **Status** to Justified, No Action Planned or Not a Defect. To hide a justified result in the next run, add the status as annotation to your source code. See “Annotate Code and Hide Known or Acceptable Results” on page 30-2.

For quick annotation, right-click the result and select **Annotate Code and Hide Result**. The option adds annotations in this format and hides the result from the results list:

```
line of code; /* polyspace Family:Result_name */
```

For details of the format, see “Annotate Code and Hide Known or Acceptable Results” on page 30-2. To unhide the hidden results, from the **Showing** menu, clear the box **Hide results justified in code**.



See Also

Related Examples

- “Specify Polyspace Compiler Options Through Eclipse Project” on page 10-7
- “Interpret Bug Finder Results in Polyspace Desktop User Interface” on page 21-2
- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 22-2
- “Filter and Group Results in Polyspace Desktop User Interface” on page 23-2

Specify Polyspace Compiler Options Through Eclipse Project

This topic describes how to configure a Polyspace analysis of Eclipse projects using Polyspace Bug Finder or Polyspace Code Prover. For the Polyspace as You Code plugin, see “Run Polyspace as You Code in Eclipse and Review Results”.

Polyspace analysis in Eclipse uses a set of default analysis options preconfigured for your coding language and operating system. For each project, you can customize the analysis options further.

- **Compiler options:** You specify the compiler that you use, the libraries that you include and the macros that are defined for your compilation.
 - If your Eclipse project directly refers to a compilation toolchain, the analysis reads the compiler options from the project.

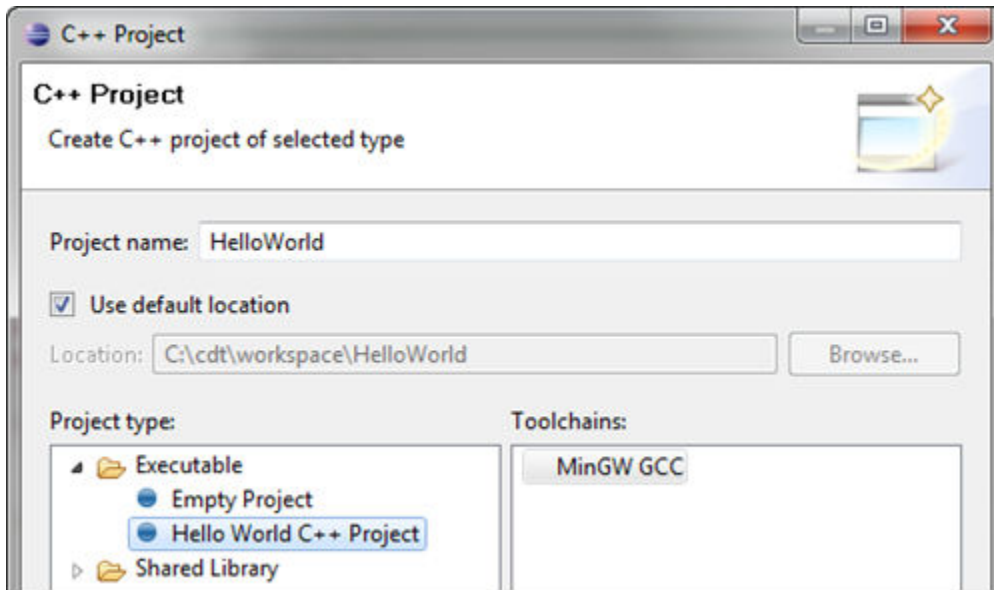
See “Eclipse Refers Directly to Your Compilation Toolchain” on page 10-7.
 - If the project refers to your compilation toolchain through a build command, the analysis cannot read the compiler options directly. Trace the build command to extract the options. Tracing a build command involves first executing the command and extracting required information from the processes executed.

See “Eclipse Uses Your Compilation Toolchain Through Build Command” on page 10-8.
- **Other options:** Through the other options, you specify which analysis results you want and how precise you want them to be. To specify these options in Eclipse, select **Polyspace > Configure Project**.

For information on how to run Polyspace from Eclipse, see “Run Polyspace Analysis on Eclipse Projects” on page 10-2.

Eclipse Refers Directly to Your Compilation Toolchain

When setting up your Eclipse project, you might be directly referring to your compilation toolchain without using a build command. For instance, you might refer to the MinGW GCC toolchain in the project setup wizard as below.



The compiler options from your Eclipse project, such as include paths and preprocessor macros, are reused for the analysis.

You cannot view the options directly in the Polyspace configuration but you can view them in your Eclipse editor. In your project properties (**Project > Properties**), in the **Paths and Symbols** node:

- See the include paths under the **Includes** tab.

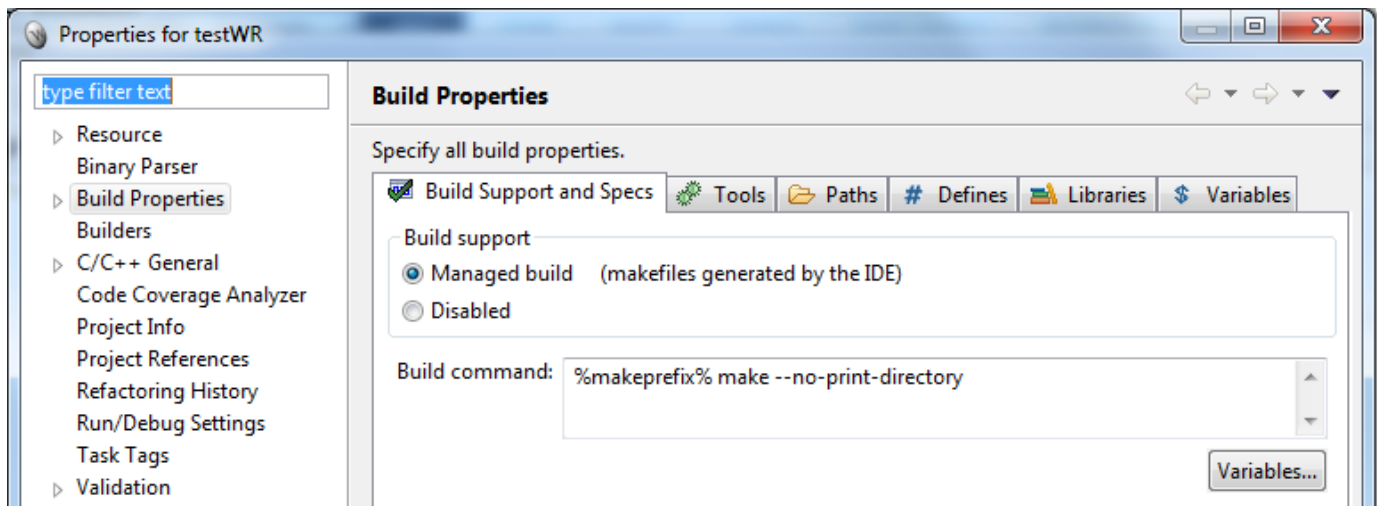
During analysis, the paths are implicitly used with the analysis option `-I`.

- See the preprocessor macros under the **Symbols** tab.

During analysis, the macros are implicitly used with the analysis option `Preprocessor definitions (-D)`.

Eclipse Uses Your Compilation Toolchain Through Build Command

When setting up your Eclipse project, instead of specifying your compilation toolchain directly, you might be specifying it through a build command. For instance, in the Wind River Workbench IDE (an Eclipse-based IDE), you might specify your build command as shown in the following figure.



If you use a build command for compilation, the analysis cannot automatically extract the compiler options. You must trace your build command.

- 1 Replace your build command with:

```
polyspaceroot\polyspace\bin\polyspace-configure.exe
-no-sources -output-project
PolyspaceWorkspace\EclipseProjects\Name\Name.psprj buildCommand
```

Here:

- *polyspaceroot* is the Polyspace installation folder.
- *polyspaceWorkspace* is the folder where your Polyspace files are stored. You specify this location on the **Project and Results Folder** tab in your Polyspace preferences (**Tools > Preferences** in the Polyspace user interface).
- *Name* is the name of your Eclipse project.
- *buildCommand* is the original build command that you want to trace.

For instance, in the preceding example, *buildCommand* is the following:

```
%makeprefix% make --no-print-directory
```

For information on the options `-output-project` and `-no-sources`, see `polyspace-configure`.

- 2 Build your Eclipse project. Perform a clean build so that files are recompiled.

For instance, select the option **Project > Clean**. Normally, the option runs your build command. With your replacement in the previous step, the option also traces the build to extract the compiler options.

- 3 Restore the original build command and restart Eclipse.

You can now run analysis on your Eclipse project. The analysis uses the compiler options that it has extracted.

See Also

Related Examples

- “Run Polyspace Analysis on Eclipse Projects” on page 10-2

Configure Polyspace as You Code

Configure Polyspace as You Code Extension in Visual Studio

Polyspace as You Code allows you to find bugs and coding rule violations while you work in your Visual Studio IDE.

After you install the Polyspace as You Code analysis engine and Visual Studio extension, configure the extension so that a Polyspace analysis runs smoothly when you save your code or explicitly start an analysis. An analysis has run smoothly if results appear as expected, either as source code markers with tooltips or in a list on the **Results List** pane.

To configure the extension, in Visual Studio:

- Select **Tools > Options** and specify the **General** settings on the **Polyspace** node. These settings apply to all projects in Visual Studio.
- Right-click a project in the Visual Studio **Solution Explorer** pane and select **Polyspace properties** to specify settings that apply only to the selected project.

All settings retain their current values when you reinstall the extension.

General Settings

Setting	Description
Analysis launch mode	Select whether Polyspace as You Code runs on each file save or explicitly. Select one of the following: <ul style="list-style-type: none"> • Automatically(default): Analysis starts on each file save. • Manually: User explicitly starts the analysis. To start an analysis, right-click in the source code or the file in the Solution Explorer, and select Run Polyspace analysis.
Polyspace as You Code installation folder	Polyspace as You Code installation folder. This field is read-only and set at the time of installation. If you see errors related to starting a Polyspace Connector, check if the folder still exists (and contains a Polyspace as You Code installation). The errors appear on the Output pane in Visual Studio.
Working directory for extension	Folder where analysis results are stored. When you start an analysis, a subfolder is created in this folder for each Visual Studio solution. Within a subfolder, a second subfolder is created per project and then another per file. For each file, a new run overwrites results of the previous run. If the analysis fails for a given file, you can check the failed subfolder for information useful for troubleshooting, such as the options given to the analysis engine. The default results folder is C:\TEMP\%USERNAME%\Polyspace.

Setting	Description
Polyspace Access URL	<p>URL of the Polyspace Access instance from which you get a baseline.</p> <p>After you obtain a baseline from Polyspace Access, subsequent runs of Polyspace as You Code allow you to distinguish between new results and results that were present in existing code (code previously uploaded to Polyspace Access).</p> <p>See also “Baseline Polyspace as You Code Results in Visual Studio” on page 11-44.</p>



Polyspace Properties for Project

Build tab

Setting	Description
Get from solution (default)	<p>If your project configuration type is <i>Application (.exe)</i>, <i>Dynamic Library (.dll)</i>, or <i>Static Library (.lib)</i>, Polyspace extracts the build options from your project when you start the analysis.</p> <p>Otherwise, before you start an analysis, click Generate Polyspace build configuration to build the Visual Studio solution, trace the build, and extract your build options. If the project configuration type is <i>Makefile</i> or <i>Utility</i>, the label for this setting lists the project type. For instance (project type: 'Utility').</p> <p>See also “Configure Polyspace as You Code to Extract Build Configuration” on page 11-27.</p>
Get from build command line	<p>Specify:</p> <ul style="list-style-type: none"> • The build command in the setting Build command line • The folder from which the build command must be launched in the setting Working directory. <p>Before you start an analysis, click Generate Polyspace build configuration to run your build command, trace your build, and extract your build options.</p> <p>See also “Configure Polyspace as You Code to Extract Build Configuration” on page 11-27.</p>

Setting	Description
<p>Get from JSON compilation database</p>	<p>Specify the path to the JSON file (typically named <code>compile_commands.json</code>) in the setting Path to JSON file.</p> <p>Before you start an analysis, click Generate Polyspace build configuration to extract the build options from the JSON compilation database that you specify.</p> <p>See also “Configure Polyspace as You Code to Extract Build Configuration” on page 11-27.</p> <p>If you use a build system generator such as CMake, you can follow this approach to set up Polyspace as You Code. For instance, if you use CMake projects in Visual Studio, CMake also allows you to generate a JSON compilation database with the commands used for building the project. You can then provide the compilation database to this setting. For an example of how to generate this JSON file, see “Create Polyspace Options File from JSON Compilation Database”.</p>
<p>Get from Polyspace build options file</p>	<p>The analysis uses manually specified options. Provide these options in the options file that you specify in the setting Build options file. See “Specify Analysis Options Manually” on page 11-29.</p>
<p>Build options file not required</p>	<p>You do not have to specify Polyspace options related to your building configuration. This option applies only to simple projects.</p> <p>The analysis uses the default Polyspace build options. You should typically provide Polyspace as You Code with the specificities of your build configuration so that the analysis runs without errors.</p>

Analysis tab

Setting	Description
<p>Checkers file</p>	<p>Path to a checkers configuration file.</p> <p>To create or edit this file, open the Checkers selection window by clicking . Enable the checkers that you want and save the file.</p> <p>To select an existing file, open the file explorer by clicking .</p> <p>See also “Configure Checkers for Polyspace as You Code in Visual Studio” on page 11-63.</p>

Setting	Description
Analysis options file	<p>Path to an options file. The options file contains one Polyspace analysis option per line. For example:</p> <pre>-D _WIN32 -termination-functions exit_handler</pre> <p>You typically do not need to specify additional options in an options file. However, in some situations, you might want to use an options file. For instance, if you want to manually specify Polyspace options related to your build command.</p> <p>See also “Options Files for Polyspace Analysis” on page 12-5.</p>
Import options from Polyspace Desktop project (*.psprj)	<p>Import the analysis options and checkers configuration file from existing Polyspace desktop project file. See “Import Analysis Options from Polyspace Desktop Project” on page 11-30.</p>

Polyspace Access tab

Setting	Description
Use baseline from Polyspace Access	<p>Specify whether to use a baseline for Polyspace results.</p> <p>If you enable this setting, specify a Project path and click Download baseline from Polyspace Access to download a baseline.</p> <p>After you download the baseline, subsequent runs of Polyspace as You Code import review information from the baseline and allow you to distinguish between new results and results that were present in existing code.</p> <p>See also “Baseline Polyspace as You Code Results in Visual Studio” on page 11-44.</p>
Show only new findings compared to the results baseline	<p>Specify whether only new results must be shown. If you select this option, results are compared with the baseline downloaded from Polyspace Access and only new results are shown.</p> <p>See also “Baseline Polyspace as You Code Results in Visual Studio” on page 11-44.</p>

Expert tab

Setting	Description
<p>Run analysis script</p>	<p>Run a script each time you save your code (or explicitly run analysis).</p> <p>The extension passes these parameters to the script:</p> <ul style="list-style-type: none"> • Path to the current file as the first argument. • Working directory for extension path as the second argument. • Polyspace as You Code installation folder as the third argument. <p>For example, this simple Windows batch script analyzes the current file, uses the default Polyspace build options, and imports the review information from a previously downloaded baseline:</p> <pre>set INSTALL_DIR=%3 set ANALYZE=%INSTALL_DIR%\polyspace\bin\polyspace-bug-finder-access.exe set SOURCES=%1 set RESULTS_FOLDER=%2 set BASELINE_DIR=%RESULTS_FOLDER%\..\..\..\baseline "%ANALYZE%" -sources %SOURCES% -import-comments %BASELINE_DIR% -results-dir %RESULTS_FOLDER% IF %ERRORLEVEL% NEQ 0 EXIT 1</pre> <p>For more on downloading a baseline and importing its review information at the command line, see “Baseline Polyspace as You Code Results on Command Line” on page 11-56.</p> <p>Use a script if, for instance, you switch between files from components that have different build configurations or you use a custom tool to setup your build environment.</p> <p>If you enable this setting, all other extension settings are ignored.</p> <hr/> <p>Note The Polyspace as You Code extension does not check the exit status of the commands in your script. Make sure your script checks exit codes (for instance by using %ERRORLEVEL%) and returns a meaningful exit status.</p> <p>Typically, the Polyspace binaries return 0 on success and a non-zero value on failure.</p>
<p>Analysis script</p>	<p>Enter the full path to a script that runs each time your run Polyspace as You Code. The script can be written in any language. On Windows, the extension supports scripting languages only for scripts that are executable from the Command Prompt.</p> <p>Depending on your Analysis launch mode setting, the script runs on each file save, or when you right-click in the source code or the file in the Solution Explorer and select Run Polyspace analysis.</p>

See Also

Related Examples

- “Generate Build Options for Polyspace as You Code Analysis in Visual Studio” on page 11-27
- “Baseline Polyspace as You Code Results in Visual Studio” on page 11-44
- “Configure Checkers for Polyspace as You Code in Visual Studio” on page 11-63
- “Run Polyspace as You Code in Visual Studio and Review Results” on page 29-2

Configure Polyspace as You Code Extension in Visual Studio Code

Polyspace as You Code allows you to find bugs and coding rule violations while you work in your Visual Studio Code editor.

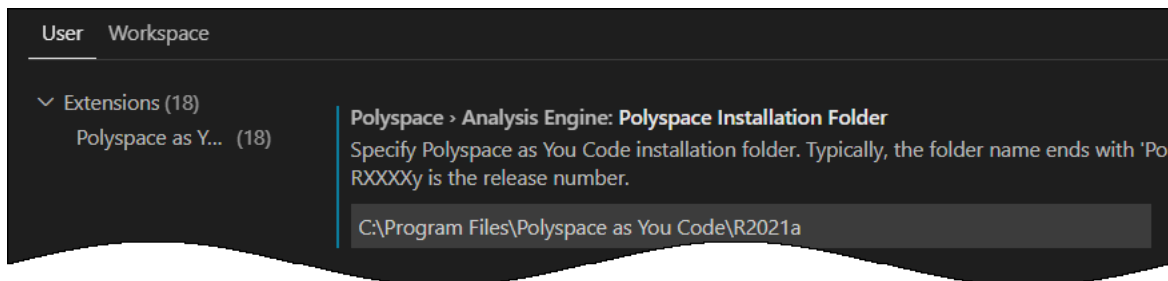
After you install the Polyspace as You Code analysis engine and Visual Studio Code extension, configure the extension so that a Polyspace analysis runs smoothly when you save your code or explicitly start an analysis. An analysis has run smoothly if results appear as expected, either as source code markers with tooltips or in a list on the **PROBLEMS** pane.


To configure the extension, in Visual Studio Code, open the settings interface by pressing **Ctrl + ,** (comma) and type `polyspace` in the settings search bar.

For each setting, you can specify a value that applies globally to all workspaces or folders that you open in the Visual Studio editor. For most of the settings, you can also override the global specification with a workspace-specific value.

- To specify global settings, enter the settings on the **User** tab.
- To override the global settings for the currently open workspace or folder, enter the settings on the **Workspace** tab or the **Remote** tab if you are using the VS Code Remote Development feature .

Unless otherwise specified, settings that are available on the **User** tab are also available on the **Workspace** or **Remote** tabs.



To reset a setting to its default value, click the  icon on the left of the setting and select **Reset Setting**. All settings retain their current values when you reinstall the extension.

Tip Type the Setting ID in the settings search bar to view only the settings related to that ID.

Analysis Engine

Setting ID: `polyspace.analysisEngine`

These settings are mandatory. For better performance on Windows, the path that you provide for these settings should not point to network drives.

Setting	Description
Polyspace Installation Folder	Root folder of the Polyspace as You Code installation, for instance, C:\Program Files\Polyspace as You Code\R2023a.
Working Directory	Folder where analysis results are stored. Each new run overwrites results of the previous run. If you do not specify a folder path, Polyspace creates a Working Directory in your system's temporary folder: <ul style="list-style-type: none"> • /tmp in Linux. • C:\users\%username%\AppData\Local\Temp in Windows.

Analysis Behavior On Save

Setting ID: polyspace.analysisOptions OnSave

By default, Polyspace as You Code adds the current file to the **Quality Monitoring** list and runs each time you save your code. You can choose to disable these automatic actions.

Setting	Description
Analysis Options: Add To Quality Monitoring On Save	Select how you add files to the Quality Monitoring list. Polyspace as You Code analyzes files only if they are added to that list. By default, Polyspace adds the current file to the list on save (Ctrl + S). Deselect this setting to add files to the Quality Monitoring list manually. To add files manually, right-click the file in the editor, the EXPLORER panel, or SOURCE CONTROL panel in the side bar.
Analysis Options: Analysis Of Files On Save	Select when Polyspace as You Code runs on files that are in the Quality Monitoring list. By default, Polyspace as You Code runs each time you save your code. Deselect this setting to run the analysis manually. You can right-click the source code or a file in the EXPLORER and select Run Polyspace Analysis (or run the command Polyspace: Run Polyspace Analysis from the Command Palette).

Analysis Setup

Setting ID: polyspace.analysisOptions.analysisSetup

You can set up a Polyspace as You Code analysis through extension settings or override extension settings and run a script instead. By default, the analysis uses extension settings.

Setting	Description
Analysis Options: Analysis Setup	Select between manual setup and script. <ul style="list-style-type: none"><li data-bbox="521 352 1464 447">• Manual Setup (default): Set up Polyspace as You Code through extension settings. Specify build-related and other options through the Manual Setup group of settings. See “Analysis Options > Manual Setup” on page 11-10.<li data-bbox="521 520 1464 583">• Script: Run a script each time you save your code (or right-click a source file and select Run Polyspace Analysis). See “Analysis Options > Script” on page 11-13.

Analysis Options > Manual Setup

Setting ID: polyspace.analysisOptions.manualSetup

Manual setup of the analysis involves specifying build options, checkers and other analysis options. Extract build options from a Visual Studio Code build task or a JSON Compilation Database file, or specify them explicitly in a build options file. Enable or disable checkers in a checkers selection window. Specify all remaining analysis options explicitly in an options file.

Setting	Description
Analysis Options > Manual Setup: Build	<p>Specification of build-related Polyspace analysis options. Options are:</p> <ul style="list-style-type: none"> • Build options file not required (default) <p>You do not have to specify Polyspace options related to building your files. This is a basic option for simple projects where the default Polyspace analysis options are sufficient to compile the files.</p> • Get from build command <p>Polyspace uses your build command to generate a build options file. Make sure that the command builds all source files in your workspace. Specify the build command in the setting Analysis Options > Manual Setup > Build Setting: Build Command.</p> <p>To generate the build options file, from the Command Palette, run Polyspace: Generate Build Options.</p> <p>See “Get Build Configuration from Build Command” on page 11-33</p> • Get from build task <p>Polyspace uses your Visual Studio build task to generate a build options file. Make sure that the build task performs a complete build of all the files in your workspace. Specify the build task name in the setting Analysis Options > Manual Setup > Build Setting: Build Task.</p> <p>To generate the build options file, from the Command Palette, run Polyspace: Generate Build Options.</p> <p>See “Get Build Configuration from Build Task” on page 11-33.</p> • Get from JSON Compilation Database file <p>Polyspace uses your JSON compilation database to generate a build options file. Specify the path to the database file (typically named <code>compile_commands.json</code>) in the setting Analysis Options > Manual Setup > Build Setting: JSON Compilation Database File.</p> <p>To generate the build options file, from the Command Palette, run Polyspace: Generate Build Options.</p> <p>See “Get Build Configuration from JSON Compilation Database” on page 11-34.</p> • Get from Polyspace build options file <p>Provide the build options in the options file that you specify in the setting Analysis Options > Manual Setup > Build Setting: Polyspace Build Options File.</p> <p>See also “Options Files for Polyspace Analysis” on page 12-5.</p>

Setting	Description
Analysis Options > Manual Setup > Build Setting: Build Command	<p>Use this setting if you choose <code>Get</code> from <code>build</code> command for the setting Analysis Options > Manual Setup: Build.</p> <p>Specify the build command name exactly as you would enter on a command-line terminal or console.</p> <p>Use a build command that performs a complete build of all files in your workspace and not an incremental build.</p> <p>See “Get Build Configuration from Build Command” on page 11-33</p>
Analysis Options > Manual Setup > Build Setting: Build Task	<p>Use this setting if you choose <code>Get</code> from <code>build</code> task for the setting Analysis Options > Manual Setup: Build.</p> <p>Specify the build task name. The build task name is the name of a command that runs when you select Terminal > Run Task. For more information on tasks, see Visual Studio Code documentation.</p> <p>Use a build task that performs a complete build of all files in your workspace and not an incremental build.</p> <p>See “Get Build Configuration from Build Task” on page 11-33.</p>
Analysis Options > Manual Setup > Build Setting: JSON Compilation Database File	<p>Use this setting if you choose <code>Get</code> from <code>JSON Compilation Database File</code> for the setting Analysis Options > Manual Setup: Build.</p> <p>Specify the full path to a database file (typically named <code>compile_commands.json</code>).</p> <p>See “Get Build Configuration from JSON Compilation Database” on page 11-34.</p> <p>If you use a build system generator such as CMake, you can follow this approach to set up Polyspace as You Code. For instance, if you use CMake projects in Visual Studio Code, CMake also allows you to generate a JSON compilation database with the commands used for building the project. You can then provide the compilation database to this setting. For an example of how to generate this JSON file, see “Create Polyspace Options File from JSON Compilation Database”.</p>
Analysis Options > Manual Setup > Build Setting: Polyspace Build Options File	<p>Use this setting if you choose <code>Get</code> from <code>Polyspace Build Options File</code> for the setting Analysis Options > Manual Setup: Build.</p> <p>Specify the full path to a Polyspace build options file. The options file is a text file with one Polyspace analysis option per line.</p> <p>See also “Options Files for Polyspace Analysis” on page 12-5.</p>
Analysis Options > Manual Setup: Checkers File	<p>Specify the full path to a checkers configuration file.</p> <p>To create this file, in the Command Palette, run Polyspace: Configure Checkers. Enable the checkers that you want and save the file.</p> <p>See also “Configure Checkers for Polyspace as You Code in Visual Studio Code” on page 11-66.</p>

Setting	Description
Analysis Options > Manual Setup: Other Analysis Options	<p>Path to an options file. The options file contains one Polyspace analysis option per line. For example:</p> <pre data-bbox="516 384 1458 443">-termination-functions exit_handler -code-behavior-specifications /usr/jdoe/util/checkerModifiers.xml</pre> <p>You typically do not need to specify additional options in an options file. However, in some situations, you might want to use an options file. For instance, you might want to modify some checkers using an XML file that you provide with the option <code>-code-behavior-specifications</code>.</p> <p>See also “Options Files for Polyspace Analysis” on page 12-5.</p>

Analysis Options > Script**Setting ID:** polyspace.analysisOptions.scriptFile

Setting	Description
Analysis Options > Script: Script File	<p>Use this setting if you choose Script for the setting Analysis Options: Analysis Setup.</p> <p>Enter the full path to a script that runs each time you run Polyspace as You Code on save or explicitly. The script can be written in any language. On Windows, the extension supports scripting languages only for scripts that are executable from the Command Prompt.</p> <p>The extension passes these parameters to the script:</p> <ul style="list-style-type: none"> • Path to the current file as the first argument. • Working Directory path as the second argument. • Polyspace Installation Folder path as the third argument. <p>For example, this simple Windows batch script analyzes the current file, uses the default Polyspace build options, and imports the review information from a previously downloaded baseline:</p> <pre>set INSTALL_DIR=%3 set ANALYZE=%INSTALL_DIR%\polyspace\bin\polyspace-bug-finder-access.exe set SOURCES=%1 set RESULTS_FOLDER=%2 set BASELINE_DIR=RESULTS_FOLDER%\..\..\..\baseline "%ANALYZE%" -sources %SOURCES% -import-comments %BASELINE_DIR% -results-dir %RESULTS_FOLDER% IF %ERRORLEVEL% NEQ 0 EXIT 1</pre> <p>For more on downloading a baseline and importing its review information at the command line, see “Baseline Polyspace as You Code Results on Command Line” on page 11-56.</p> <p>Use a script if, for instance, you switch between files from components that have different build configurations or you use a custom tool to setup your build environment.</p> <p>If you enable this setting, all other extension settings are ignored.</p> <p>Note The Polyspace as You Code extension does not check the exit status of the commands in your script. Make sure your script checks exit codes (for instance by using %ERRORLEVEL%) and returns a meaningful exit status.</p> <p>Typically, the Polyspace binaries return 0 on success and a non-zero value on failure.</p>

Baseline

Setting ID: polyspace.baseline

Set these options if you want to compare your local results against a baseline from Polyspace Access. After you obtain a baseline from Polyspace Access, subsequent runs of Polyspace as You Code allow you to distinguish between new results and results that were present in existing code. See also “Baseline Polyspace as You Code Results in Visual Studio Code” on page 11-48.

Setting	Description
Baseline: Polyspace Access Login	Specify the user name that you use to log in to Polyspace Access. Later, when you run Polyspace: Download Baseline in the Command Palette , you are prompted for the password that corresponds to this user name.
Baseline: Polyspace Access Url	Specify the Polyspace Access URL, for instance <code>https://example.access-server:9443</code> .
Baseline: Project	Specify the path of a project on Polyspace Access that you use as baseline. To download the baseline, from the Command Palette , run Polyspace: Download Baseline . See also “Baseline Polyspace as You Code Results in Visual Studio Code” on page 11-48.
Baseline: Show Baseline Information	Enable or disable the use of information from the baseline run: <ul style="list-style-type: none"> • Show local findings only (default) When you run an analysis, Polyspace does not use information from the baseline run. You see only local findings in the PROBLEMS pane. • Show local findings and baseline info When you run an analysis, Polyspace imports review information from the baseline run. Results that are already justified in the baseline run are suppressed in the PROBLEMS pane. • Show new findings only When you run an analysis, Polyspace imports review information from the baseline. Results that are already present in the baseline run are suppressed in the PROBLEMS pane. <p>Use this setting to focus only on new findings.</p> <p>See also “Baseline Polyspace as You Code Results in Visual Studio Code” on page 11-48.</p>

Justification Catalog

Setting ID: `polyspace.justification.catalog`

Use this setting if your team or organization has a predefined set of comments that they use to justify results. You can store these justifications in a catalog file and associate one or more justification with a specific result or result family.

If you use the Polyspace syntax to annotate a result in the Visual Studio Code editor, justifications that you store in the catalog for that result are available in a dropdown when you start typing the annotation comment. For details of the Polyspace syntax, see “Annotation Syntax Details” on page 30-4.

To create a justification catalog, see “Use a Justification Catalog to Autocomplete Annotations in Polyspace as You Code plugins” on page 29-27.

Setting	Description
Justification: Catalog	Specify the full file path of the JSON catalog.

Other Settings

Setting ID: polyspace.otherSettings

Additional settings to configure the debugging mode and the port the extension uses to communicate with the analysis engine.

Setting	Description
Other Settings: Debug Mode	<p>Enable or disable debugging.</p> <ul style="list-style-type: none"> Disabled(default) <p>Show only errors, warnings, and information messages such as start and end of analysis in the OUTPUT pane.</p> Enabled <p>Show all debugging information in the OUTPUT pane. Enable this setting to troubleshoot issues with the Polyspace as You Code extension.</p>
Other Settings: Headers Extensions	<p>Specify the extensions that Polyspace as You Code should treat as header files instead of source files.</p> <p>If you configure Polyspace to add files to the Quality Monitoring list on save and you make edits to a file with one of the specified header extensions, Polyspace does not add that file to the Quality Monitoring list on save.</p> <p>Polyspace analyzes and reports findings in a header file if you analyze a file that includes the header file or if you explicitly start an analysis of the header file. The header file is analyzed even if you do not specify the extension of the file in this setting.</p> <p>See also “HEADERS view” on page 29-7.</p>
Other Settings: Help Improve Polyspace as You Code	<p>Enable or disable the sharing of user experience information with MathWorks. Polyspace uses this information to improve the Polyspace as You Code extension.</p> <p>This setting is available only on the User tab.</p>

Setting	Description
Other Settings: Port	<p>Specify the port number that the Polyspace as You Code extension uses on startup to establish an internal connection with the analysis engine.</p> <p>Use this setting if, for instance, your machine is configured with a firewall and you want to specify an open port in the firewall.</p> <p>By default, port 0 is specified and Polyspace queries your system for an available port and uses whichever port your system returns.</p> <p>If you run multiple instances of the Polyspace as You Code extension, specify a different port for each instance.</p> <p>If you change this setting, you might need to reload Visual Studio Code. If you let Polyspace obtain a port number automatically (port 0), the extension might connect on a different port when you reload Visual Studio Code and you might need to rerun the command <code>Polyspace: Generate Build Options</code>.</p>

Configure Polyspace as You Code for Remote Development

The Polyspace as You Code extension supports the Visual Studio Code remote development feature. See [VS Code Remote Development](#).

With remote development, you can run a Polyspace as You Code analysis from your local machine on code that you develop and edit on a remote machine. To enable remote development, see [Remote tutorials](#).

When you enable remote development:

- The Polyspace as You Code extension is disabled on your local machine. You use the Polyspace as You Code extension on the remote machine to run the analysis.
- You see an additional **Remote** tab in the Polyspace extension settings. You configure the extension on the remote machine in this tab.
- If you set **Analysis Options > Manual Setup: Build** to `Get from build task` or `Get from build command`, check that your build completes successfully before you run the `Polyspace: Generate Build Options` command.

Note On Windows, if any of your project files or folders on the remote machine are on a network drive, provide the UNC path for that network drive. The Polyspace as You Code extension cannot resolve the path of a network drive that is mapped to a drive letter when you enable remote development and the `Polyspace: Generate Build Options` command might fail.

See Also

Related Examples

- “Generate Build Options for Polyspace as You Code Analysis in Visual Studio Code” on page 11-32
- “Baseline Polyspace as You Code Results in Visual Studio Code” on page 11-48

- “Configure Checkers for Polyspace as You Code in Visual Studio Code” on page 11-66
- “Run Polyspace as You Code in Visual Studio Code and Review Results” on page 29-6

Configure Polyspace as You Code Plugin in Eclipse

This topic describes how to configure the Polyspace as You Code plugin in Eclipse. For Polyspace desktop products such as Polyspace Bug Finder, see “Bug Finder Analysis Based on Eclipse Projects”.

Polyspace as You Code allows you to find bugs and coding rule violations while you work in your Eclipse-based IDE.

After you install the Polyspace as You Code analysis engine and Eclipse plugin, configure the plugin so that a Polyspace analysis runs smoothly when you save your code or explicitly start an analysis. An analysis has run smoothly if results appear as expected, either as source code markers with tooltips or in a list on the **Polyspace Problems** view.

To configure the Eclipse plugin, go to **Window > Preferences** and select the **Polyspace as You Code** node in the **Preferences** window. You can also open the **Preferences** window by pressing **Alt +W+P**. Under the **Polyspace as You Code** node, select:

- **Analysis** to configure the analysis options for the projects.
- **Baseline** to configure the use of analysis results you download from Polyspace Access as a baseline.

To save your configuration, click **Apply and Close**, or click **Apply** to save your changes and continue making edits to the configuration. To restore default configuration values, click **Restore Defaults**. All settings retain their current values when you reinstall the plugin.

Polyspace as You Code Node

Installation and results folders

Setting	Description
Polyspace installation folder	Root folder of the Polyspace as You Code installation, for instance, C:\Program Files\Polyspace as You Code\R2023a.
Working directory	Folder where analysis results are stored. Each new run overwrites results of the previous run. The default working directory, is stored under your system temporary folder, and typically contains <code>polyspace_eclipse</code> in the folder name: <ul style="list-style-type: none"> • Windows: %temp% folder, for instance C:\Users\jsmith\AppData\Local\Temp. • Linux: /tmp folder.

Polyspace justification

Use this setting if your team or organization has a predefined set of comments that they use to justify results. You can store these justifications in a catalog file and associate one or more justification with a specific result or result family.

If you use the Polyspace syntax to annotate a result, the annotation comment is auto-filled with the justifications that you store in the catalog for that result. For details of the Polyspace syntax, see “Annotation Syntax Details” on page 30-4.

To create a justification catalog, see “Use a Justification Catalog to Autocomplete Annotations in Polyspace as You Code plugins” on page 29-27.

Setting	Description
Justification catalog	Specify the full file path of the justification catalog JSON file.

Other

Setting	Description
Debug Mode	<p>Enable or disable debugging.</p> <ul style="list-style-type: none"> <input type="checkbox"/> (default) — The Console view shows only errors, warnings, and information messages such as start and end of analysis. <input checked="" type="checkbox"/> — The Console view shows all debugging information. Enable this setting to troubleshoot issues with the Eclipse plugin.
Help Improve Polyspace as You Code	<p>Enable (default) or disable the sharing of user experience information with MathWorks.</p> <ul style="list-style-type: none"> <input checked="" type="checkbox"/> (default) — Polyspace shares user experience information with MathWorks and uses this information to improve the Eclipse plugin. <input type="checkbox"/> — Polyspace does not share user experience information. <p>Polyspace</p>
Port	<p>Specify the port number that the Polyspace as You Code plugin uses on startup to establish an internal connection with the analysis engine.</p> <p>Use this setting if, for instance, your machine is configured with a firewall and you want to specify an open port in the firewall.</p> <p>By default, port 0 is specified and Polyspace queries your system for an available port and uses whichever port your system returns.</p> <p>If you run multiple instances of Eclipse which access different workspaces, specify a different port for each instance.</p> <p>If you change this setting, you might need to restart your Eclipse IDE. If you let Polyspace obtain a port number automatically (port 0), the plugin might connect on a different port when you restart your Eclipse IDE and you might need to regenerate your build options.</p>

Analysis Node

Plugin Behavior on Save

Specify whether the plugin performs certain actions when you save your edits

Setting	Description
Add To Quality Monitoring list on save	<p>Select how you add files to the Quality Monitoring list. Polyspace as You Code analyzes files only if they are added to that list.</p> <ul style="list-style-type: none"> <input checked="" type="checkbox"/> (default) — Polyspace adds the current file to the list on save (Ctrl + S). <input type="checkbox"/> — add files to the Quality Monitoring list manually. To add files manually, right-click the file in the editor or the Project Explorer view.
Start analysis on save	<p>Select when Polyspace as You Code runs on files that are in the Quality Monitoring list.</p> <ul style="list-style-type: none"> <input checked="" type="checkbox"/> — Polyspace as You Code runs each time you save your code. (Ctrl + S). <input type="checkbox"/> (default) — Run the analysis manually. Right-click the source code or a file in the Project Explorer and select Run Polyspace Analysis.

Build Configuration

Setting	Description
Analysis Setup	<p>Select between manual setup and script.</p> <ul style="list-style-type: none"> Manual Setup (default): Set up Polyspace as You Code through extension settings. Specify build-related and other options through the Manual Setup: Manual setup group of settings. Script: Run a script each time you save your code (or right-click a source file and select Run Polyspace Analysis).

Analysis Setup — Script

Run a script each time you save your code or explicitly run analysis.

The plugin passes these parameters to the script:

- Path to the current file as the first argument.
- **Working directory** path as the second argument.
- **Polyspace installation folder** path as the third argument.

For example, this simple Windows batch script analyzes the current file, uses the default Polyspace build options, and imports the review information from a previously downloaded baseline:

```
set INSTALL_DIR=%3
set ANALYZE=%INSTALL_DIR%\polyspace\bin\polyspace-bug-finder-access.exe
set SOURCES=%1
set RESULTS_FOLDER=%2
set BASELINE_DIR=%RESULTS_FOLDER%\..\..\baseline
```

```
"%ANALYZE%" -sources %SOURCES% -import-comments %BASELINE_DIR% -results-dir %RESULTS_FOLDER%
IF %ERRORLEVEL% NEQ 0 EXIT 1
```

For more on downloading a baseline and importing its review information at the command line, see “Baseline Polyspace as You Code Results on Command Line” on page 11-56.

Use a script if, for instance, you switch between files from components that have different build configurations or you use a custom tool to setup your build environment.




If you enable this setting, all other plugin settings are ignored.

Note The Polyspace as You Code plugin does not check the exit status of the commands in your script. Make sure your script checks exit codes (for instance by using %ERRORLEVEL%) and returns a meaningful exit status.

Typically, the Polyspace binaries return 0 on success and a non-zero value on failure.

Setting	Description
Script file	Enter the full path to a script that runs each time you run Polyspace as You Code. The script can be written in any language. On Windows, the extension supports scripting languages only for scripts that are executable from the Command Prompt . Depending on how you configure the plugin behavior on save, the script runs on each file save, or when you right-click in the source code or the file in the Project Explorer and select Run Polyspace Analysis .

Analysis Setup — Manual

Setting	Description
Build	<ul style="list-style-type: none"> <li data-bbox="519 352 1162 386">• Build options file not required (default) <p data-bbox="555 411 1354 474">You do not have to specify Polyspace options related to your build configuration. This is a basic option for simple projects.</p> <p data-bbox="555 499 651 533">More...</p> <p data-bbox="555 558 1455 655">The analysis uses the default Polyspace build options. So that the analysis runs without errors, you typically provide Polyspace as You Code with the specificities of your build configuration.</p> <ul style="list-style-type: none"> <li data-bbox="519 667 906 701">• Get from build command <p data-bbox="555 726 1435 789">The analysis traces the build command that you specify and generates a build options file.</p> <p data-bbox="555 814 651 848">More...</p> <p data-bbox="555 873 1464 1012">Specify the build command in setting Build Command, for instance make -B and click  in the Configuration view of the Polyspace as You Code perspective. See “Get Build Configuration from Build Command” on page 11-37.</p> <ul style="list-style-type: none"> <li data-bbox="519 1024 1036 1058">• Get from Eclipse C/C++ project <p data-bbox="555 1083 1451 1146">The analysis extracts the build configuration from the Eclipse project and generates a build options file.</p> <p data-bbox="555 1171 651 1205">More...</p> <p data-bbox="555 1230 1377 1306">Click  in the Configuration view of the Polyspace as You Code perspective.</p> <p data-bbox="555 1331 1360 1365">See “Get Build Configuration from Eclipse Project” on page 11-37.</p> <ul style="list-style-type: none"> <li data-bbox="519 1377 1179 1411">• Get from JSON Compilation Database file <p data-bbox="555 1436 1471 1533">The analysis extracts the build configuration from the JSON compilation database that you specify and generates a build options file. See “Get Build Configuration from JSON Compilation Database” on page 11-38.</p> <p data-bbox="555 1558 651 1591">More...</p> <p data-bbox="555 1617 1442 1755">Specify the full path to the JSON file (typically named <code>compile_commands.json</code>) in the JSON Compilation Database file setting and click  in the Configuration view of the Polyspace as You Code perspective.</p> <p data-bbox="555 1780 1468 1902">If you use a build system generator such as CMake, you can follow this approach to set up Polyspace as You Code. For instance, if you use CMake projects in Eclipse, CMake also allows you to generate a JSON compilation database with the commands used for building the project. You can then</p>

Setting	Description
	<p>provide the compilation database to this setting. For an example of how to generate this JSON file, see “Create Polyspace Options File from JSON Compilation Database”.</p> <ul style="list-style-type: none"> • <code>Get from Polyspace build options file</code> <p>When you select this setting, you provide an options file that you generate or fill in manually with all the necessary build options. Specify the full path of the options file in the setting Polyspace Build Options File.</p>
Build Command	<p>Specify the build command name exactly as you would enter on a command-line terminal or console.</p> <p>Use a build command that performs a complete build of all files in your workspace and not an incremental build.</p> <p>Fill in this setting if you select <code>Get from build command</code> from the Build dropdown menu. See “Get Build Configuration from Build Command” on page 11-37.</p>
JSON Compilation Database file	<p>Specify the full path to a database file (typically named <code>compile_commands.json</code>).</p> <p>Use a build command that performs a complete build of all files in your workspace and not an incremental build.</p> <p>Fill in this setting if you select <code>Get from JSON Compilation Database file</code> from the Build dropdown menu. See “Create Polyspace Options File from JSON Compilation Database”.</p>
Polyspace Build Options File	<p>Specify the full path to a Polyspace build options file. The options file is a text file with one Polyspace analysis option per line.</p> <p>Fill in this setting if you select <code>Get from Polyspace build options file</code> from the Build dropdown menu. See “Options Files for Polyspace Analysis” on page 12-5.</p>
Checkers file	<p>Path to a checkers configuration file.</p> <p>To create this file, click the Configure Checkers icon in the Configuration view of the Polyspace as You Code perspective.. Enable the checkers that you want and save the file.</p> <p>See also “Configure Checkers for Polyspace as You Code in Eclipse” on page 11-60.</p>

Setting	Description
Other Analysis Options	<p>Path to an options file. The options file contains one Polyspace analysis option per line. For example:</p> <pre>-D _WIN32 -termination-functions exit_handler</pre> <p>You typically do not need to specify additional options in an options file. However, in some situations, you might want to use an options file. For instance, if you want to manually specify Polyspace options related to your build command, select Build options file not required for Build setting and enter the options in an options file.</p> <p>See also “Options Files for Polyspace Analysis” on page 12-5.</p>

Baseline Node

Use this settings to select whether you import review information from a baseline run into your analysis, and to configure the connection to a Polyspace Access Server from which you download the baseline.

Setting	Description
Show Baseline Information	<p>Enable or disable the use of information from the baseline run:</p> <ul style="list-style-type: none"> • Show local findings only (default) <p>When you run an analysis, Polyspace does not use information from the baseline run. You see only local findings in the Polyspace Problems view.</p> • Show local findings and baseline info <p>When you run an analysis, Polyspace imports review information from the baseline run. Results that are already justified in the baseline run are suppressed in the Polyspace Problems pane.</p> • Show new findings only <p>When you run an analysis, Polyspace imports review information from the baseline. Results that are already present in the baseline run are suppressed in the Polyspace Problems pane.</p> <p>Use this setting to focus only on new findings.</p> <p>See also “Baseline Polyspace as You Code Results in Eclipse” on page 11-53.</p>
Polyspace Access URL	<p>URL of the Polyspace Access instance from which you get a baseline.</p> <p>After you obtain a baseline from Polyspace Access, subsequent runs of Polyspace as You Code allow you to distinguish between new results and results that were present in existing code.</p> <p>See also “Baseline Polyspace as You Code Results in Eclipse” on page 11-53.</p>

Setting	Description
Polyspace Access Login	Specify the user name that you use to log in to Polyspace Access. Later, when you click the Download Baseline icon in the Baseline view of the Polyspace as You Code perspective, you are prompted for the password that corresponds to this user name.
Project path	Path of project in Polyspace Access Project Explorer that you get the baseline from.

See Also

Related Examples

- “Generate Build Options for Polyspace as You Code Analysis in Eclipse” on page 11-37
- “Baseline Polyspace as You Code Results in Eclipse” on page 11-53
- “Configure Checkers for Polyspace as You Code in Eclipse” on page 11-60
- “Run Polyspace as You Code in Eclipse and Review Results” on page 29-15

Generate Build Options for Polyspace as You Code Analysis in Visual Studio

Polyspace as You Code checks the source code file that is currently active in your Visual Studio IDE for bugs and coding standards violations.

So that the analysis runs without errors, provide Polyspace as You Code with the specificities of your build configuration, such as data type sizes and compiler macro definitions. To provide your build configuration information, you can:

- Configure Polyspace as You Code to extract the build configuration information from your Visual Studio solution, build command, or JSON compilation database. Note that running `polyspace-configure` on a build command involves first executing the command and gathering information from the processes executed. On the other hand, `polyspace-configure` can simply read all required information from a JSON compilation database or even a Visual Studio solution in some cases (see details later).
- Manually specify analysis options that emulate your build configuration in an options file. See “Options Files for Polyspace Analysis” on page 12-5.
- Import the analysis options from a Polyspace desktop product project file.

Configure Polyspace as You Code to Extract Build Configuration

To extract your build configuration information from the Visual Studio solution, build command, or JSON compilation database:

- 1 Right-click a project in the Visual Studio **Solution Explorer** pane and select **Polyspace properties**.
- 2 Select the appropriate build configuration option on the **Build** tab. See “Configure Polyspace as You Code Extension in Visual Studio” on page 11-2.

The build configuration option that you select applies only to the selected project.

Polyspace extracts the build information and generates an options file that the Polyspace as You Code analysis engine uses in subsequent analyses.

- The file contains analysis options that emulate your build configuration. Make sure that the build completes successfully before you use this file .
- The generated options file is stored in the `.polyspace-configure` folder under the `workingDirectory/projectName` folder or one of its subfolders.

The `workingDirectory` path is the **Working directory for extension** folder path that you specify in the **General** options of the Polyspace extension. The `projectName` is the name of the project that contains the files you are currently analyzing.

Build Option	Description
<p>Get from solution</p>	<ul style="list-style-type: none"> • If your project configuration type is <i>Application (.exe)</i>, <i>Dynamic Library (.dll)</i>, or <i>Static Library (.lib)</i>, when you start an analysis, Polyspace extracts the build options from your project and generates an options file. If you make changes to your project, Polyspace updates the options file when you start the next analysis. • If your project configuration type is <i>Makefile</i> or <i>Utility</i>, or if you do not use the <i>cl.exe</i> compiler to compile your code, before the analysis starts, Polyspace builds the Visual Studio solution and traces the build to extract your build options and generate an options file. If you make changes to your project, update the generated options file before you start the next analysis. See “Update Generated Build Options File” on page 11-29. <p>Polyspace builds your solution and generates an options file only if:</p> <ul style="list-style-type: none"> • You start an analysis and Polyspace cannot find a generated options file in the <code>.polyspace-configure</code> folder for the project that contains the currently analyzed file. • You explicitly generate an options file by selecting Generate Polyspace build configuration from your project context menu in the Solution Explorer.
<p>Get from build command line</p>	<ul style="list-style-type: none"> • Specify your build command in the Build command line field. The build command that you specify must perform a full build. For instance: <pre>cl /W1 hello.c main.c</pre> • Specify the full path of the folder where Polyspace runs the build command in the Working directory field. For instance: <pre>C:\Projects\HelloWorld</pre> <p>Before the analysis starts, Polyspace runs your build command, traces the build to extract the configuration information, and generates an options file. If you make changes to your project, update the generated options file before you start the next analysis. See “Update Generated Build Options File” on page 11-29.</p> <p>Polyspace runs your build command and generates an options file only if:</p> <ul style="list-style-type: none"> • You start an analysis and Polyspace cannot find a generated options file in the <code>.polyspace-configure</code> folder for the project that contains the currently analyzed file. • You explicitly generate an options file by selecting Generate Polyspace build configuration from your project context menu in the Solution Explorer.

Build Option	Description
Get from JSON compilation database	<p>If your build system supports the generation of a JSON compilation database file, use this setting. The file contains compiler calls for all the translation units in your project. See JSON compilation database.</p> <p>To extract your build configuration information from the JSON compilation database:</p> <ol style="list-style-type: none"> 1 Generate a JSON compilation database file. For an example of how to generate this file, see “Create Polyspace Options File from JSON Compilation Database”. <p>If you use a JSON compilation database that was not generated on your local machine, make sure that the paths listed in the file are accessible from the location where you run Polyspace as You Code.</p> <ol style="list-style-type: none"> 2 Specify the full path to the JSON compilation database file that you generated in step 1 in the Path to JSON file field. <p>Before the analysis starts, Polyspace extracts the build configuration information from the compilation database and generates an options file. If you make changes to your project, update the generated options file before you start the next analysis. See “Update Generated Build Options File” on page 11-29.</p> <p>Polyspace generates an options file only if:</p> <ul style="list-style-type: none"> • You start an analysis and Polyspace cannot find a generated options file in the <code>.polyspace-configure</code> folder for the project that contains the currently analyzed file. • You explicitly generate an options file by selecting Generate Polyspace build configuration from your project context menu in the Solution Explorer.

Update Generated Build Options File

If you make changes to your build configuration, for instance if you add a source file to your project or workspace or rename an existing file, update the generated options file to reflect those changes. Before you update the options file, make sure that your build completes successfully with the new configuration.

To update the options file, from the project context menu in the **Solution Explorer**, select **Generate Polyspace build configuration**.

If you extract your build information from a JSON compilation database file, regenerate the compilation database before you update the build options file.

See also “Troubleshoot Failed Analysis or Unexpected Results in Polyspace as You Code” on page 11-81.

Specify Analysis Options Manually

Use this setting if:

- You know the details of your build system and you want to specify the Polyspace analysis options that emulate your build configuration in an options file. See “Options Files for Polyspace Analysis” on page 12-5.

For a list of available analysis options, see “Complete List of Polyspace Bug Finder Analysis Engine Options”.

- You reuse a Polyspace options file that you or someone else on your team has configured for your build system.

If you reuse an options file that was not configured or generated on your local machine, make sure that the paths listed in the file are accessible from the location where you run Polyspace as You Code.

To specify an analysis options file:

- 1 Right-click a project in the Visual Studio **Solution Explorer** pane and select **Polyspace properties**.
- 2 Select **Get from build options file** on the **Build** tab. See “Configure Polyspace as You Code Extension in Visual Studio” on page 11-2.
- 3 Specify the full path to the options file in the **Build options file** field.

The Polyspace as You Code analysis engine uses the specified options file in subsequent analyses.

If you make changes to your build configuration, edit the options file to reflect those changes. See “Specify Target Environment and Compiler Behavior” on page 13-2.

Import Analysis Options from Polyspace Desktop Project

If you configure an analysis in the Polyspace desktop product, you can use the information from the resulting Polyspace desktop PSPRJ file to configure your Polyspace as You Code analysis.

To import the analysis options from a Polyspace desktop PSPRJ file:

- 1 Right-click a project in the Visual Studio **Solution Explorer** pane and select **Polyspace properties**.
- 2 Select **Build options file not required** on the **Build** tab.
- 3 On the **Analysis** tab, click **Import from Polyspace desktop project** and select the PSPRJ file that you import from.

Polyspace generates an options file and an XML checkers activation file on page 11-63, and populates the **Checkers file** and **Analysis options file** field on the **Analysis** tab. The Polyspace as You Code analysis engine uses these files in subsequent analyses.

If you make changes to your build configuration, edit the options file to reflect those changes. See “Specify Target Environment and Compiler Behavior” on page 13-2.

See Also

Related Examples

- “Configure Polyspace as You Code Extension in Visual Studio” on page 11-2

- “Configure Checkers for Polyspace as You Code in Visual Studio” on page 11-63
- “Baseline Polyspace as You Code Results in Visual Studio” on page 11-44

Generate Build Options for Polyspace as You Code Analysis in Visual Studio Code

Polyspace as You Code checks the source code file that is currently active in your Visual Studio Code editor for bugs and coding standards violations.

So that the analysis runs without errors, provide Polyspace as You Code with the specificities of your build configuration, such as data type sizes and compiler macro definitions. To provide your build configuration information, you can:

- Configure Polyspace as You Code to extract the build configuration information from your build task or build command, or JSON compilation database. Note that running `polyspace-configure` on a build task or command involves first executing the task or command and gathering information from the processes executed. On the other hand, `polyspace-configure` can simply read all required information from a JSON compilation database.
- Manually specify analysis options that emulate your build configuration in an options file. See “Options Files for Polyspace Analysis” on page 12-5.
- Import the analysis options from a Polyspace desktop product project file.

Configure Polyspace as You Code to Extract Build Configuration

To extract your build configuration information from the build task, build command, or JSON compilation database:

- 1 Open the Visual Studio Code settings by pressing **Ctrl+,** (comma).

Enter `polyspace.analysisoptions` in the settings search bar and set **Polyspace > Analysis Options: Analysis Setup** to `Manual setup`.

- 2 Set the appropriate **Polyspace > Analysis Options > Manual Setup: Build** options and fill out the corresponding **Build Setting** field.

See “Configure Polyspace as You Code Extension in Visual Studio Code” on page 11-8.

- 3 Open the **Command Palette (Ctrl+Shift+P)** and enter `Polyspace: Generate Build Options`.

Note On Windows, if you enable remote development and if any of your project files or folders on the remote machine are on a network drive, provide the UNC path for that network drive. See “Configure Polyspace as You Code for Remote Development” on page 11-17. The Polyspace as You Code extension cannot resolve the path of a network drive that is mapped to a drive letter when you enable remote development and the `Polyspace: Generate Build Options` command might fail.

Polyspace extracts the build information and generates an options file that the Polyspace as You Code analysis engine uses in subsequent analyses. The file contains analysis options that emulate your build configuration.

The generated options file is stored in the `.polyspace-configure` folder under the `workingDirectory/projectName` folder or one of its subfolders.

The *workingDirectory* path is the **Polyspace > Analysis Engine: Result Folder** path that you specify in the Polyspace as You Code extension settings. The *projectName* is the name of the top-level folder in the **EXPLORER** that contains the files that you are currently analyzing.

Get Build Configuration from Build Task

Visual Studio Code enables you to define tasks so that you can run an external tool from your code editor. See Integrate with External Tools via Tasks.

If you define a custom task that calls your compiler to perform a full build of your project, Polyspace can extract your build configuration from this build task.

- 1 Open the Visual Studio Code settings by pressing **Ctrl+,** (comma).

Enter `polyspace.analysisoptions` in the settings search bar.

- 2 Set these **Polyspace > Analysis Options** settings to the values listed in the table.

Setting	Value
Analysis Setup	Manual setup
Manual Setup: Build	Get from build task
Manual Setup > Build Setting: Build Task	Specify the name of the build task. This corresponds to the "label" field of the task definition in the <code>tasks.json</code> file. The task that you specify must perform a full build. Polyspace supports the use of only these Visual Studio Code predefined variables in task definitions: <ul style="list-style-type: none"> • <code>\${workspaceFolder}</code> • <code>\${workspaceFolderBasename}</code>

- 3 Open the **Command Palette (Ctrl+Shift+P)** and enter **Polyspace: Generate Build Options**.

Polyspace runs the build command specified by the task, traces the build to extract the configuration information, and generates an options file.

Get Build Configuration from Build Command

To extract your build configuration information from your build command:

- 1 Open the Visual Studio Code settings by pressing **Ctrl+,** (comma).

Enter `polyspace.analysisoptions` in the settings search bar.

- 2 Set these **Polyspace > Analysis Options** settings to the values listed in the table.

Setting	Value
Analysis Setup	Manual setup
Manual Setup: Build	Get from build command

Setting	Value
Manual Setup > Build Setting: Build Command	Specify your build command, for instance: <code>gcc -g -o helloworld hello.c main.c</code> The command that you specify must perform a full build

- 3 Open the **Command Palette (Ctrl+Shift+P)** and enter Polyspace: Generate Build Options.

Polyspace runs your build command, traces the build to extract the configuration information, and generates an options file.

Get Build Configuration from JSON Compilation Database

If your build system supports the generation of a JSON compilation database file, use this setting. The file contains compiler calls for all the translation units in your project. See JSON compilation database.

To extract your build configuration information from the JSON compilation database:

- 1 Generate a JSON compilation database file. For an example of how to generate this file, see “Create Polyspace Options File from JSON Compilation Database”.

If you use a JSON compilation database that was not generated on your local machine, make sure that the paths listed in the file are accessible from the location where you run Polyspace as You Code.

- 2 Open the Visual Studio Code settings by pressing **Ctrl+,** (comma).

Enter `polyspace.analysisoptions` in the settings search bar.

- 3 Set these **Polyspace > Analysis Options** settings to the values listed in the table.

Setting	Value
Analysis Setup	Manual setup
Manual Setup: Build	Get from JSON Compilation Database file
Manual Setup > Build Setting: JSON Compilation Database File	Specify the full path to the file that you generated in step 1. The file is typically named <code>compile_commands.json</code> .

- 4 Open the **Command Palette (Ctrl+Shift+P)** and enter Polyspace: Generate Build Options.

Polyspace extracts the build configuration information from the compilation database and generates an options file.

Update Generated Build Options File

If you make changes to your build configuration, for instance if you add a source file to your project or workspace or rename an existing file, update the generated options file to reflect those changes. Before you update the options file, make sure that your build completes successfully with the new configuration.

To update the options file, Open the **Command Palette (Ctrl+Shift+P)** and enter **Polyspace: Generate Build Options**.

If you extract your build information from a JSON compilation database file, regenerate the compilation database before you update the build options file.

See also “Troubleshoot Failed Analysis or Unexpected Results in Polyspace as You Code” on page 11-81.

Specify Analysis Options Manually

Use this setting if:

- You know the details of your build system and you want to specify the Polyspace analysis options that emulate your build configuration in an options file. See “Options Files for Polyspace Analysis” on page 12-5.

For a list of available analysis options, see “Complete List of Polyspace Bug Finder Analysis Engine Options”.

- You reuse a Polyspace options file that you or someone else on your team has configured for your build system.

If you reuse an options file that was not configured or generated on your local machine, make sure that the paths listed in the file are accessible from the location where you run Polyspace as You Code.

To specify an analysis options file:

- 1 Open the Visual Studio Code settings by pressing **Ctrl+,** (comma).

Enter `polyspace.analysisoptions` in the settings search bar.

- 2 Set these **Polyspace > Analysis Options** settings to the values listed in the table.

Setting	Value
Analysis Setup	Manual setup
Manual Setup: Build	Get from Polyspace build options file
Manual Setup > Build Setting: Polyspace Build Options File	Specify the full path to the Polyspace options file.

The Polyspace as You Code analysis engine uses the specified options file in subsequent analyses.

If you make changes to your build configuration, edit the options file to reflect those changes. See “Specify Target Environment and Compiler Behavior” on page 13-2.

Import Analysis Options from Polyspace Desktop Project

If you configure an analysis in the Polyspace desktop product, you can use the information from the resulting Polyspace desktop PSPRJ file to configure your Polyspace as You Code analysis.

To import the analysis options from a Polyspace desktop PSPRJ file, open a terminal in Visual Studio Code and enter this command:

```
polyspace-checkers-selection -import-options-from-psprj pathToPsprjFile
```

The `polyspace-checkers-selection` binary is available under the `polyspace/bin` folder in your Polyspace as You Code installation folder. If you did not add this installation folder to your `PATH` environment variable, include the full path of the binary to execute this command.

The `pathToPsprjFile` path is the full path of the PSPRJ file.

Polyspace generates an options file (`analysis_options.txt`) and an XML checkers activation file on page 11-66 (`checkers_activation_file.xml`). The generated files are stored in the `import` folder in the same location as the PSPRJ file.

To complete the configuration of the Polyspace as You Code analysis:

- 1 Open the Visual Studio Code settings by pressing **Ctrl+,** (comma).

Enter `polyspace.analysisoptions` in the settings search bar.

- 2 Set these **Polyspace > Analysis Options** settings to the values listed in the table.

Setting	Value
Analysis Setup	Manual setup
Manual Setup: Build	Build options file not required This setting ignores the file specified in the Build Setting: Polyspace Build Options File field.
Manual Setup: Checkers File	Full file path of <code>checkers_activation_file.xml</code>
Manual Setup: Other Analysis Options	Full file path of <code>analysis_options.txt</code>

The Polyspace as You Code analysis engine uses these files in subsequent analyses.

If you make changes to your build configuration, edit the options file to reflect those changes. See “Specify Target Environment and Compiler Behavior” on page 13-2.

See Also

Related Examples

- “Configure Polyspace as You Code Extension in Visual Studio Code” on page 11-8
- “Configure Checkers for Polyspace as You Code in Visual Studio Code” on page 11-66
- “Baseline Polyspace as You Code Results in Visual Studio Code” on page 11-48

Generate Build Options for Polyspace as You Code Analysis in Eclipse

This topic describes how to configure the Polyspace as You Code plugin in Eclipse. For Polyspace desktop products such as Polyspace Bug Finder, see “Bug Finder Analysis Based on Eclipse Projects”.

Polyspace as You Code checks the source code of the file that is currently active in your Eclipse IDE for bugs and coding standards violations.

So that the analysis runs without errors, provide Polyspace as You Code with the specificities of your build configuration, such as data type sizes and compiler macro definitions. To provide your build configuration information, you can:

- Configure Polyspace as You Code to extract the build configuration information from your Eclipse project, build command, or JSON compilation database. Note that running `polyspace-configure` on a build command involves first executing the command and gathering information from the processes executed. On the other hand, `polyspace-configure` can simply read all required information from an Eclipse project or JSON compilation database.
- Manually specify analysis options that emulate your build configuration in an options file. See “Options Files for Polyspace Analysis” on page 12-5.
- Import the analysis options from a Polyspace desktop product project file.

Configure Polyspace as You Code to Extract Build Configuration


To extract your build configuration information from the Eclipse project, build command, or JSON compilation database:

- 1 Go to **Window > Preferences** and select the **Polyspace as You Code** node.
- 2 Select the appropriate **Build** option on the **Analysis** node. See “Configure Polyspace as You Code Plugin in Eclipse” on page 11-19.

Get Build Configuration from Eclipse Project

To extract your build configuration information from your Eclipse project:

- 1 Go to **Window > Preferences** and select the **Polyspace as You Code** node.
- 2 On the **Analysis** node and select **Get from Eclipse C/C++ project** from the **Build** dropdown menu. See “Configure Polyspace as You Code Plugin in Eclipse” on page 11-19.

Click  in the **Configuration** view of the **Polyspace as You Code** perspective to extract the build options from your project toolchain. The Polyspace analysis engine uses those options in the subsequent analysis.

To view the details of the toolchain configuration:

- 1 Select a project in the **Project Explorer** and go to **Project > Properties**.
- 2 Under the **C/C++ General** node, select **Paths and symbols** and **Preprocessor Include Paths, Macros, etc**.


Get Build Configuration from Build Command

To extract your build configuration information from your build command:

- 1 Go to **Window > Preferences** and select the **Polyspace as You Code** node.
- 2 On the **Analysis** node select **Get from build command** from the **Build** dropdown menu. See “Configure Polyspace as You Code Plugin in Eclipse” on page 11-19.

The build command that you specify must perform a full build. For instance:

```
gcc -g -o helloworld hello.c main.c
```

- 3 Click  in the **Configuration** view of the **Polyspace as You Code** perspective.

Polyspace runs your build command, traces the build to extract the configuration information, and generates an options file. The Polyspace as You Code analysis engine uses the generated options file in subsequent analyses.

The generated options file is stored in the `.polyspace-configure` folder under the `workingDirectory/projectName` folder or one of its subfolders.

Here, *workingDirectory* is the **Working directory** path that you specify on the **Polyspace as You Code** node and *projectName* is the name of the project that contains the files you are currently analyzing.


Get Build Configuration from JSON Compilation Database

If your build system supports the generation of a JSON compilation database file, use this setting. The file contains compiler calls for all the translation units in your project. See JSON compilation database.

To extract your build configuration information from the JSON compilation database:

- 1 Generate a JSON compilation database file. For an example of how to generate this file, see “Create Polyspace Options File from JSON Compilation Database”.

If you use a JSON compilation database that was not generated on your local machine, make sure that the paths listed in the file are accessible from the location where you run Polyspace as You Code.

- 2 Go to **Window > Preferences** and select the **Polyspace as You Code** node.
- 3 On the **Analysis** node select **Get from JSON Compilation Database file** from the **Build** dropdown menu and specify the full path to the JSON compilation database file that you generated in step 1. See “Configure Polyspace as You Code Plugin in Eclipse” on page 11-19.
- 4 Click  in the **Configuration** view of the **Polyspace as You Code** perspective.


Polyspace extracts the build configuration information from the compilation database and generates an options file. The Polyspace as You Code analysis engine uses the generated options file in subsequent analyses.

The generated options file is stored in the `.polyspace-configure` folder under the `workingDirectory/projectName` folder or one of its subfolders.

Here, *workingDirectory* is the **Working directory** path that you specify on the **Polyspace as You Code** node and *projectName* is the name of the project that contains the files you are currently analyzing.

Update Generated Build Options File

If you make changes to your build configuration, for instance if you add a source file to your project or workspace or rename an existing file, update the generated options file to reflect those changes. Before you update the options file, make sure that your build completes successfully with the new configuration.

To update the options file, click  in the **Configuration** view of the **Polyspace as You Code** perspective.

If you extract your build information from a JSON compilation database file, regenerate the compilation database before you update the build options file.

See also “Troubleshoot Failed Analysis or Unexpected Results in Polyspace as You Code” on page 11-81.

Specify Analysis Options Manually

Use this setting if:

- You know the details of your build system and you want to specify the Polyspace analysis options that emulate your build configuration in an options file. See “Options Files for Polyspace Analysis” on page 12-5.

For a list of available analysis options, see “Complete List of Polyspace Bug Finder Analysis Engine Options”.

- You reuse a Polyspace options file that you or someone else on your team has configured for your build system.

If you reuse an options file that was not configured or generated on your local machine, make sure that the paths listed in the file are accessible from the location where you run Polyspace as You Code.

To specify an analysis options file:

- 1 Go to **Window > Preferences** and select the **Polyspace as You Code** node.
- 2 On the **Analysis** node select **Get from Polyspace build options file** from the **Build** dropdown menu and specify the full path to the options file. See “Configure Polyspace as You Code Plugin in Eclipse” on page 11-19.

The Polyspace as You Code analysis engine uses the options file that you specify in subsequent analyses.

If you make changes to your build configuration, edit the options file to reflect those changes. See “Specify Target Environment and Compiler Behavior” on page 13-2.

Import Analysis Options from Polyspace Desktop Project

If you configure an analysis in the Polyspace desktop product, you can use the information from the resulting Polyspace desktop PSPRJ file to configure your Polyspace as You Code analysis.

To import the analysis options from a Polyspace desktop PSPRJ file, open a terminal in Eclipse and enter this command:

```
polyspace-checkers-selection -import-options-from-psprj pathToPsprjFile
```

The `polyspace-checkers-selection` binary is available under the `polyspace/bin` folder in your Polyspace as You Code installation folder. If you did not add this installation folder to your `PATH` environment variable, include the full path of the binary to execute this command.

The *pathToPsprjFile* path is the full path of the PSPRJ file.

Polyspace generates an options file (`analysis_options.txt`) and an XML checkers activation file on page 11-66 (`checkers_activation_file.xml`). The generated files are stored in the `import` folder in the same location as the PSPRJ file.

To complete the configuration of the Polyspace as You Code analysis:

- 1 Go to **Window > Preferences** and select the **Polyspace as You Code** node.
- 2 On the **Analysis** node select `Get from Polyspace build options file` from the **Build** dropdown. See “Configure Polyspace as You Code Plugin in Eclipse” on page 11-19.
- 3 Specify the path of the generated options file `analysis_options.txt` in the **Other Analysis Options** field.
- 4 Specify the path of the generated checkers activation file `checkers_activation_file.xml` in the **Checkers File** field.

The Polyspace as You Code analysis engine uses these files in subsequent analyses.

If you make changes to your build configuration, edit the options file to reflect those changes. See “Specify Target Environment and Compiler Behavior” on page 13-2.

See Also

Related Examples

- “Configure Polyspace as You Code Plugin in Eclipse” on page 11-19
- “Configure Checkers for Polyspace as You Code in Eclipse” on page 11-60
- “Baseline Polyspace as You Code Results in Eclipse” on page 11-53

Generate Build Options for Polyspace as You Code Analysis at the Command Line

Polyspace as You Code checks your code for bugs and coding standards violations while you work in your IDE or code editor.

So that the analysis runs without errors, provide Polyspace as You Code with the specificities of your build configuration, such as data type sizes and compiler macro definitions. To provide your build configuration information, you can:

- Use the `polyspace-configure` binary to extract the build configuration information from your build command or JSON compilation database. Note that running `polyspace-configure` on a build command involves first executing the command and gathering information from the processes executed. On the other hand, `polyspace-configure` can simply read all required information from a JSON compilation database.
- Manually specify analysis options that emulate your build configuration in an options file. See “Options Files for Polyspace Analysis” on page 12-5.
- Import the analysis options from a Polyspace desktop product project file.

Use `polyspace-configure` to Generate Build Options File

The `polyspace-configure` binary enables you to extract the build configuration information from a build command or a JSON compilation database file. The binary uses the extracted information to generate a build options file which contains a set of options that emulate your build configuration.

`polyspace-configure` is available with your Polyspace as You Code installation, in the `polyspaceAsYouCodeRoot/polyspace/bin` folder, where `polyspaceAsYouCodeRoot` is your Polyspace as You Code installation folder.

Get Build Configuration from Build Command

To extract the build configuration information from your build command, provide a build command that performs a full build. For instance, if you use `make` on Linux to build your project, use this command:

```
polyspace-configure \
-no-sources -allow-override \
-output-options-file path/To/buildOptions.txt \
-merge-common-options make -B
```

Polyspace runs your build command, traces the build to extract the configuration information, and generates `buildOptions.txt` inside `path/To`. For more information about the `polyspace-configure` options, see `polyspace-configure`.

Use the generated options file in subsequent analyses of source files from your project. For instance:

```
polyspace-bug-finder-access -sources file.c -options-file path/To/buildOptions.txt
```

Get Build Configuration from JSON Compilation Database

If your build system supports the generation of a JSON compilation database file, use this workflow.

The compilation database file contains compiler calls for all the translation units in your project. See JSON compilation database.

To extract your build configuration information from the JSON compilation database:

- 1 Generate a JSON compilation database file. For an example of how to generate this file, see “Create Polyspace Options File from JSON Compilation Database”. The generated file is typically named `compile_commands.json`.

If you use a JSON compilation database that was not generated on your local machine, make sure that the paths listed in the file are accessible from the location where you run Polyspace as You Code.

- 2 Pass the compilation database file to `polyspace-configure`. For instance:

```
polyspace-configure \  
-no-sources -allow-overwrite \  
-output-options-file path/To/buildOptions.txt \  
-merge-common-options \  
-compilation-database otherPath/To/compile_commands.json
```

Polyspace extracts the build configuration information from the compilation database and generates an options file. For more information about the `polyspace-configure` options, see `polyspace-configure`

Use the generated options file in subsequent analyses of source files from your project. For instance:

```
polyspace-bug-finder-access -sources file.c -options-file path/To/buildOptions.txt
```

Update Generated Build Options File

If you make changes to your build configuration, for instance if you add a source file to your project or workspace or rename an existing file, update the generated options file to reflect those changes. Before you update the options file, make sure that your build completes successfully with the new configuration.

To update the options file, rerun the command that you used to generate the file and specify the same set of options you used.

If you extract your build information from a JSON compilation database file, regenerate the compilation database before you update the build options file.

Specify Analysis Options Manually

Use this workflow if:

- You know the details of your build system and you want to specify the Polyspace analysis options that emulate your build configuration in an options file. See “Options Files for Polyspace Analysis” on page 12-5.

For a list of available analysis options, see “Complete List of Polyspace Bug Finder Analysis Engine Options”.

- You reuse a Polyspace options file that you or someone else on your team has configured for your build system.

If you reuse an options file that was not configured or generated on your local machine, make sure that the paths listed in the file are accessible from the location where you run Polyspace as You Code.

If you make changes to your build configuration, edit the options file to reflect those changes. See “Specify Target Environment and Compiler Behavior” on page 13-2.

Import Analysis Options from Polyspace Desktop Project

If you configure an analysis in the Polyspace desktop product, you can use the information from the resulting Polyspace desktop PSPRJ file to configure your Polyspace as You Code analysis.

To import the analysis options from a Polyspace desktop PSPRJ file, use this command:

```
polyspace-checkers-selection -import-options-from-psprj pathToPsprjFile
```

The `polyspace-checkers-selection` binary is available under the `polyspace/bin` folder in your Polyspace as You Code installation folder.

The *pathToPsprjFile* path is the full path of the PSPRJ file.

Polyspace generates an options file (`analysis_options.txt`) and an XML checkers activation file on page 11-70 (`checkers_activation_file.xml`). The generated files are stored in the `import` folder in the same location as the PSPRJ file.

Use the generated options file and checkers activation file in subsequent analyses of source files from your project. For instance:

```
polyspace-bug-finder-access -sources file.c \  
-options-file path/To/import/analysis_options.txt \  
-checkers-activation-file path/To/import/checkers_activation_file.xml
```

If you make changes to your build configuration, edit the options file (`analysis_options.txt`) to reflect those changes. See “Specify Target Environment and Compiler Behavior” on page 13-2.

See Also

`polyspace-configure` | `polyspace-bug-finder-access`

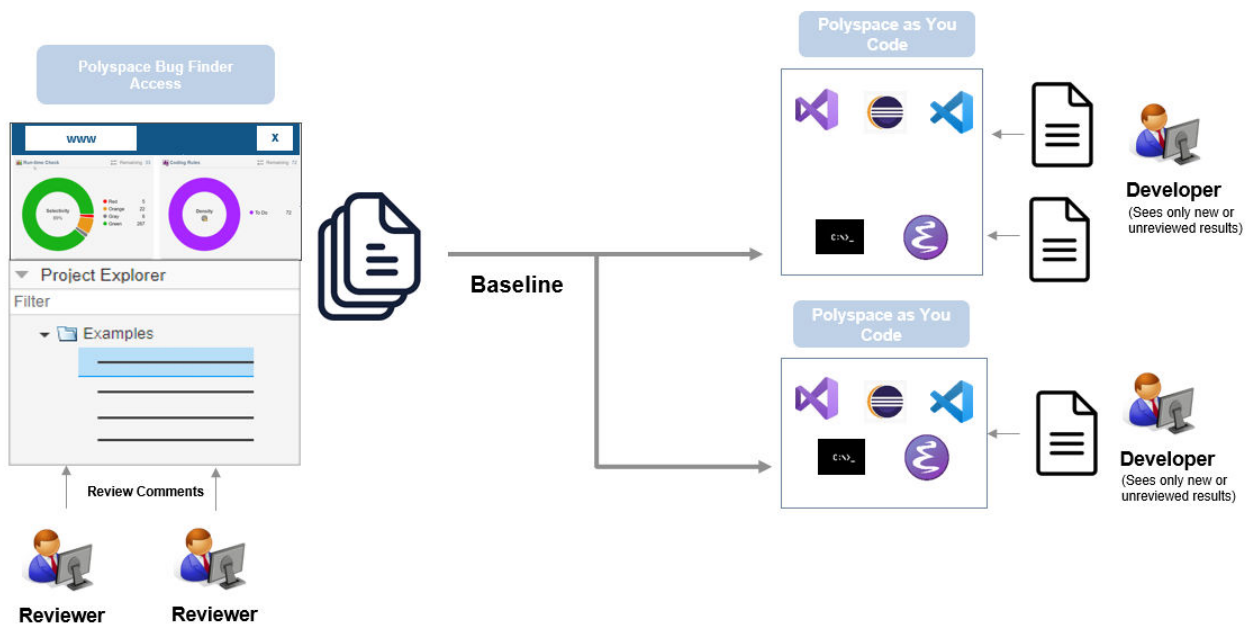
Related Examples

- “Options Files for Polyspace Analysis” on page 12-5
- “Configure Checkers for Polyspace as You Code at the Command Line” on page 11-70
- “Baseline Polyspace as You Code Results on Command Line” on page 11-56

Baseline Polyspace as You Code Results in Visual Studio

For more efficient bug fixing, you can baseline the results of a Polyspace as You Code analysis with previous results. When you baseline the results, you compare them against the results of a previous run and focus on new results only or on unreviewed results only. You baseline Polyspace as You Code results using previous Polyspace Bug Finder Server results that you download from Polyspace Access.

Note To keep using the most up-to-date baseline information, make sure that you periodically run the command to download results from Polyspace Access. See “Step 3: Download Baseline” on page 11-47.



What Baselined Results Look Like

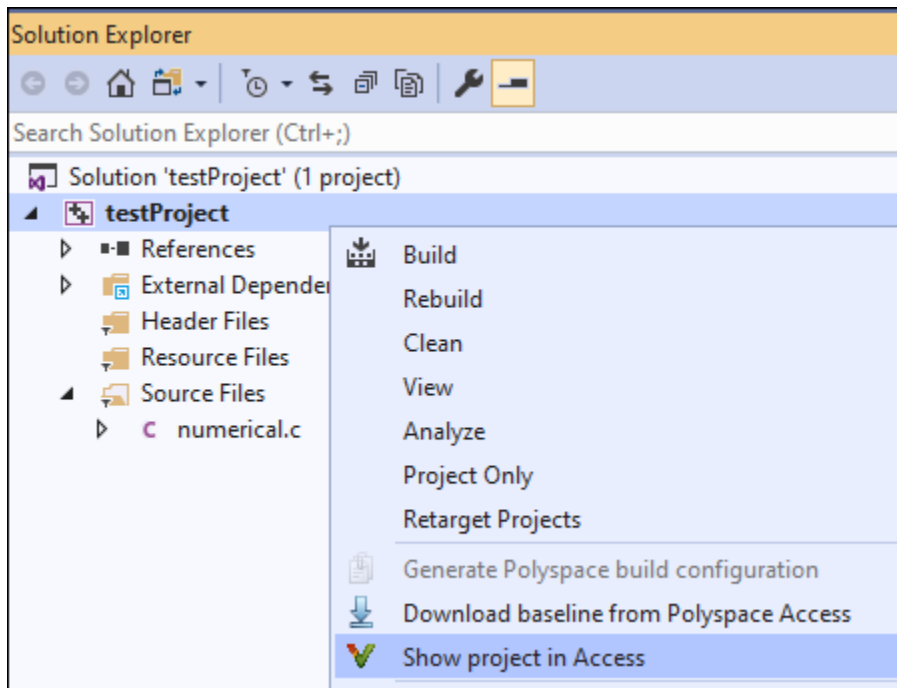
If you baseline Polyspace as You Code results using Polyspace Bug Finder Server results that you downloaded from Polyspace Access, you can see the following benefits:

- Results that have a justified **Status** on Polyspace Access (**No Action Planned**, **Justified**, or **Not a Defect**) are no longer shown.
- Results that have a non-justified **Status** on Polyspace Access carry over all review information to the **Polyspace Results List** pane in Visual Studio. If a result is reviewed in Polyspace Access and marked as such, *one of the following* is true:
 - The **Status** is different from **Unreviewed**.
 - The **Severity** is different from **Unset**.
 - The **Comment** is not blank.

For instance, the fact that the **Float division by zero** defect has associated review information indicates that the defect is also present in the baseline. In Polyspace Access, the defect has been reviewed and assigned a **Status** of **To fix**.

Polyspace Results List											
Family	New	Ln	Ch	Check	Type	Group	Information	Status	Severity	Comment	
○	No	314	16	Invalid use of standard library integer routine	Defect	Numerical	Impact: High	Unreviewed	Unset		
○	No	312	16	Invalid use of standard library integer routine	Defect	Numerical	Impact: High	Unreviewed	Unset		
○	No	185	12	Float conversion overflow	Defect	Numerical	Impact: High	Unreviewed	Unset		
○	No	112	12	Integer conversion overflow	Defect	Numerical	Impact: High	Unreviewed	Unset		
○	No	290	17	Absorption of float operand	Defect	Numerical	Impact: High	Unreviewed	Unset		
○	No	70	16	Float division by zero	Defect	Numerical	Impact: High	To fix	Unset		
○	No	36	14	Integer division by zero	Defect	Numerical	Impact: High	Unreviewed	Unset		

You can also open the Polyspace Access project used as baseline in a web browser. In Visual Studio, right-click the project on the **Solution Explorer** pane and select **Show project in Access**.



- If you specify the Polyspace as You Code extension setting **Show only new findings compared to the results baseline**, you see only results that are new in the current run. That way, you can focus only on results that explicitly occurred because of the changes you made since the last Polyspace Server run.

Baselining Steps

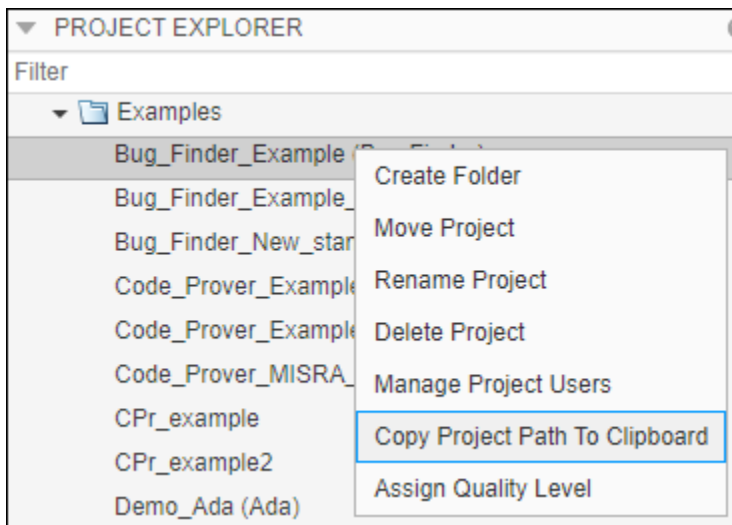
To use Polyspace Bug Finder Server results as baseline for a Polyspace as You Code analysis, follow the steps below. Once a baseline is downloaded, if you choose to point to the baseline, each subsequent run, whether on file save or on-demand, uses the baseline.

Step 1: Identify Project to Use as Baseline

First, identify a project in Polyspace Access that you want to use as baseline. The project must contain results of a Polyspace Bug Finder Server analysis on files that you will analyze in Polyspace as You Code.

Copy the path to the project for use in the Visual Studio Code extension settings. To copy this path:

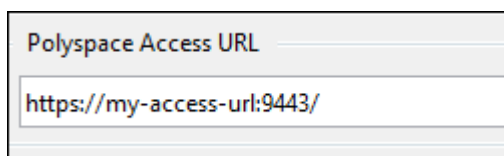
- 1 Open the Polyspace Access web interface in a web browser.
- 2 On the **Project Explorer** pane, right-click the project and select **Copy Project Path to Clipboard**.



Step 2: Refer to Project from Polyspace as You Code

Next, refer to the Polyspace Access project from the Polyspace as You Code extension settings in Visual Studio.

- 1 Go to **Tools > Options** and select the **Polyspace** node.
- 2 Specify on the **General** node the **Polyspace Access URL**. This is the URL of the server that hosts Polyspace Access. For instance, `https://my-access-url:9443/`.



- 3 Right-click the project on the **Solution Explorer** pane and select **Polyspace properties**.

Specify these settings on the **Polyspace Access** tab:

- **Use baseline from Polyspace Access:** Select this option to use the project on Polyspace Access as baseline.
- **Project path:** The path to the project in Polyspace Access that you want to use as baseline. You previously copied this path from the Polyspace Access web interface.

- **Show only new findings compared to the results baseline:** Select this option to suppress results that are already present in the project in Polyspace Access.

Step 3: Download Baseline

Explicitly download the Polyspace Access result to use as baseline.

- 1 The first time you configure this settings, click **Download baseline from Polyspace Access** to download a baseline.

To download an updated baseline later, right-click the project on the **Solution Explorer** pane and select **Download baseline from Polyspace Access**.

- 2 Enter the username and password that you use to log in to Polyspace Access. The baseline download begins.

To follow the progress of download, select **View > Output** and from the dropdown on the top, select **Polyspace**. Wait for the message:

Baseline downloaded successfully for Access project *ProjectName*

After download, subsequent runs use the baseline. To disable baseline usage, disable the extension setting **Use baseline from Polyspace Access**.

See Also

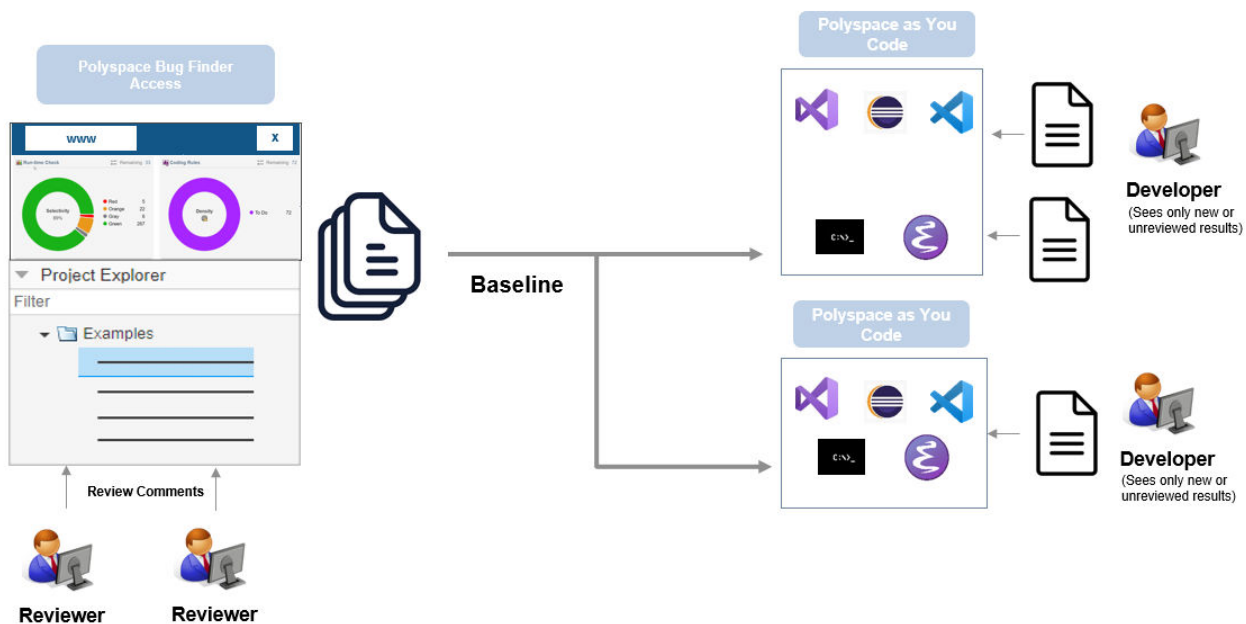
More About

- “Configure Polyspace as You Code Extension in Visual Studio” on page 11-2
- “Run Polyspace as You Code in Visual Studio and Review Results” on page 29-2

Baseline Polyspace as You Code Results in Visual Studio Code

For more efficient bug fixing, you can baseline the results of a Polyspace as You Code analysis using previous results. When you baseline the results, you compare them against the results of a previous run and focus on new results only or on unreviewed results only. You baseline Polyspace as You Code results using previous Polyspace Bug Finder Server results that you download from Polyspace Access.

Note To keep using the most up-to-date baseline information, make sure that you periodically run the command to download results from Polyspace Access. See “Step 3: Download Baseline” on page 11-51.

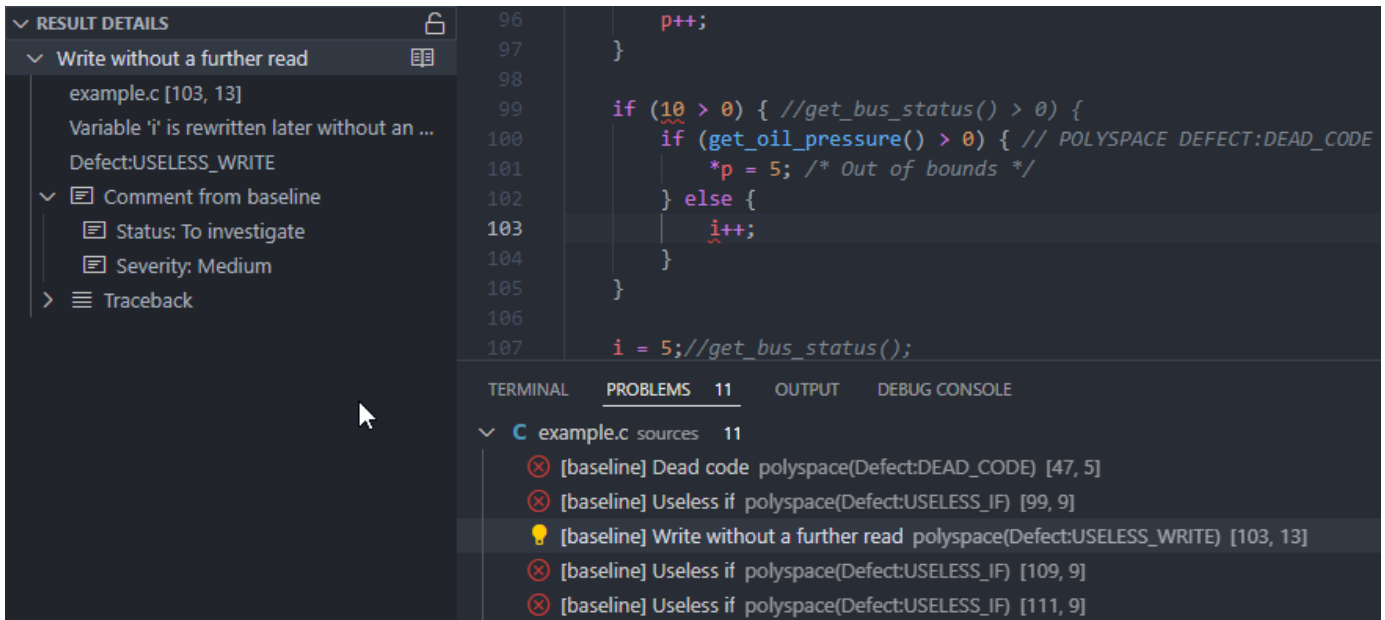



What Baselined Results Look Like

If you baseline Polyspace as You Code results using Polyspace Bug Finder Server results that you downloaded from Polyspace Access, you can see the following benefits:

- Results that have a justified **Status** on Polyspace Access (**No Action Planned**, **Justified**, or **Not a Defect**) are no longer shown.
- Results that have a non-justified **Status** on Polyspace Access carry over all review information to the **RESULT DETAILS** view.

For instance, the **Write without a further read** defect below is also present in the baseline. In Polyspace Access, the defect has been reviewed and assigned a **Status** of **To investigate** and a **Severity** of **Medium**. This review information appears in the **Comment from baseline** section in the **RESULT DETAILS** view. The word [baseline] also appears next to the result in the **PROBLEMS** panel.



If a Polyspace as You Code result also appears in the baseline, you can open the project that you use as baseline in the Polyspace Access interface. Click  in the **BASELINE** view. See “Open Additional Polyspace Views” on page 29-7.

You can also open the project on Polyspace Access by running **Polyspace: Open Project in Access Web** from the **Command Palette**.

- If you specify Show local findings only in the **Baseline: Show Baseline Information** Polyspace as You Code extension setting, you see only results that are new in the current run. That way, you can only focus on results that explicitly occurred because of the changes you made since the last Polyspace Server run.

Baselining Steps

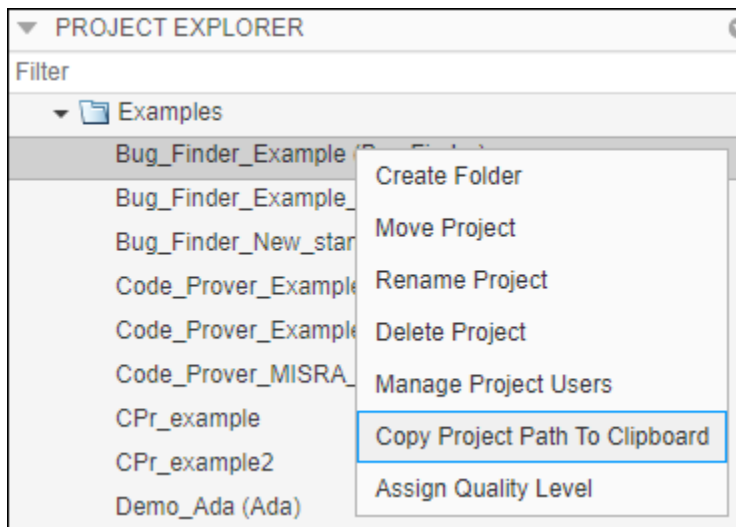
To use Polyspace Bug Finder Server results as baseline for a Polyspace as You Code analysis, follow the steps below. Once a baseline is downloaded, if you choose to point to the baseline, each subsequent run, whether on file save or on-demand, uses the baseline.

Step 1: Identify Project to Use as Baseline

First, identify a project in Polyspace Access that you want to use as baseline. The project must contain results of a Polyspace Bug Finder Server analysis on files that you will analyze in Polyspace as You Code.

Copy the path to the project for use in the Visual Studio Code extension settings. To copy this path:

- 1 Open the Polyspace Access web interface in a web browser.
- 2 On the **Project Explorer** pane, right-click the project and select **Copy Project Path to Clipboard**.



Step 2: Refer to Project from Polyspace as You Code

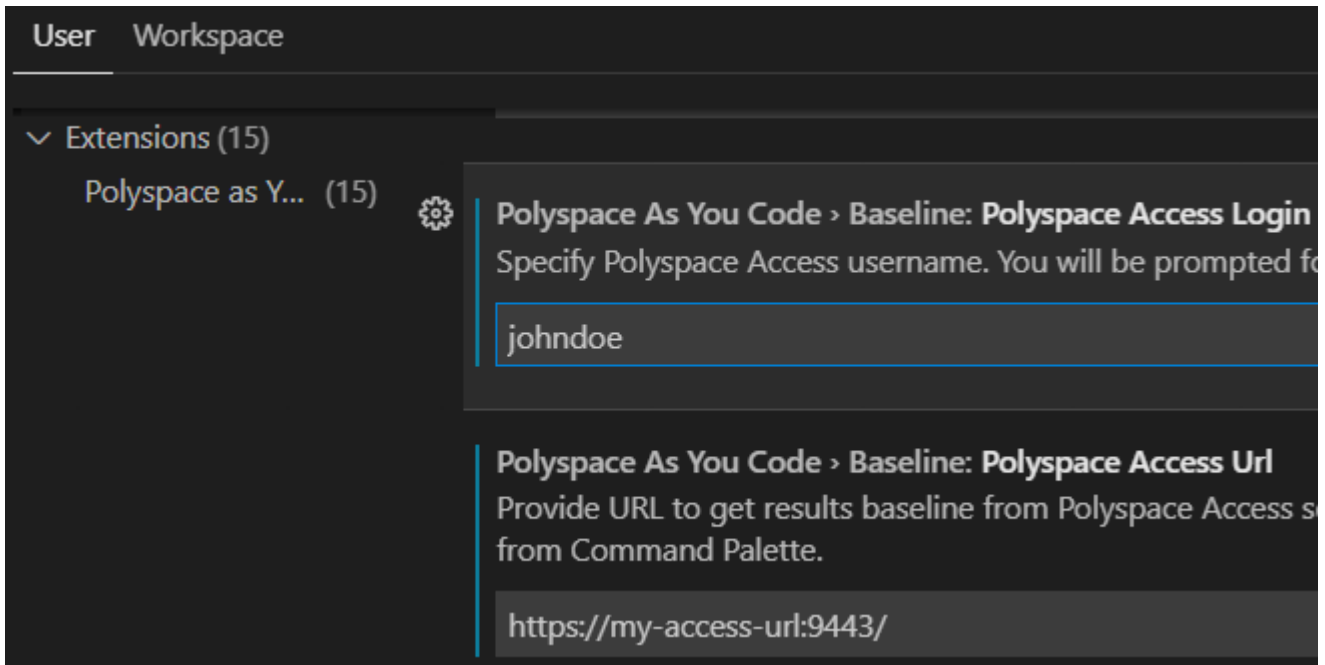
Next, refer to the Polyspace Access project from the Polyspace as You Code extension settings in Visual Studio Code.

- 1 Open the extension settings.

To open the baseline settings, press **Ctrl + ,** (comma) and type `polyspace.baseline` in the settings search bar.

- 2 Specify these settings on the **User** tab:

- **Baseline: Polyspace Access Login:** The username that you use to log in to Polyspace Access.
- **Baseline: Polyspace Access Url:** The URL of the server that hosts Polyspace Access. For instance, `https://my-access-url:9443/`.




3 Specify these settings on the **Workspace** tab:

- **Baseline: Project:** The path to the project in Polyspace Access that you want to use as baseline. You previously copied this path from the Polyspace Access web interface.
- **Baseline: Show Baseline Information:** Select an option from the dropdown.
 - Show local findings only — Do not use the baseline.
 - Show local findings and baseline info — After you download a baseline, findings that are already justified in the baseline run are suppressed in the **PROBLEMS** panel.
 - Show new findings only — After you download a baseline, findings that are already present in the baseline run are suppressed in the **PROBLEMS** panel. Use this option to focus on only new findings in your file.

Step 3: Download Baseline

Explicitly download the Polyspace Access result to use as baseline.

- 1** Click  in the **BASELINE** view or open the command palette (**Ctrl + Shift + P**) and select **Polyspace: Download Baseline**.
- 2** Enter the password that you use to log in to Polyspace Access. The baseline download begins.

You can view the progress of the download in the **BASELINE** view. See “Open Additional Polyspace Views” on page 29-7.

After download, subsequent runs use the baseline. To disable baseline usage, select Show local findings only in the **Baseline: Show Baseline Information** drop down menu.

See Also

More About

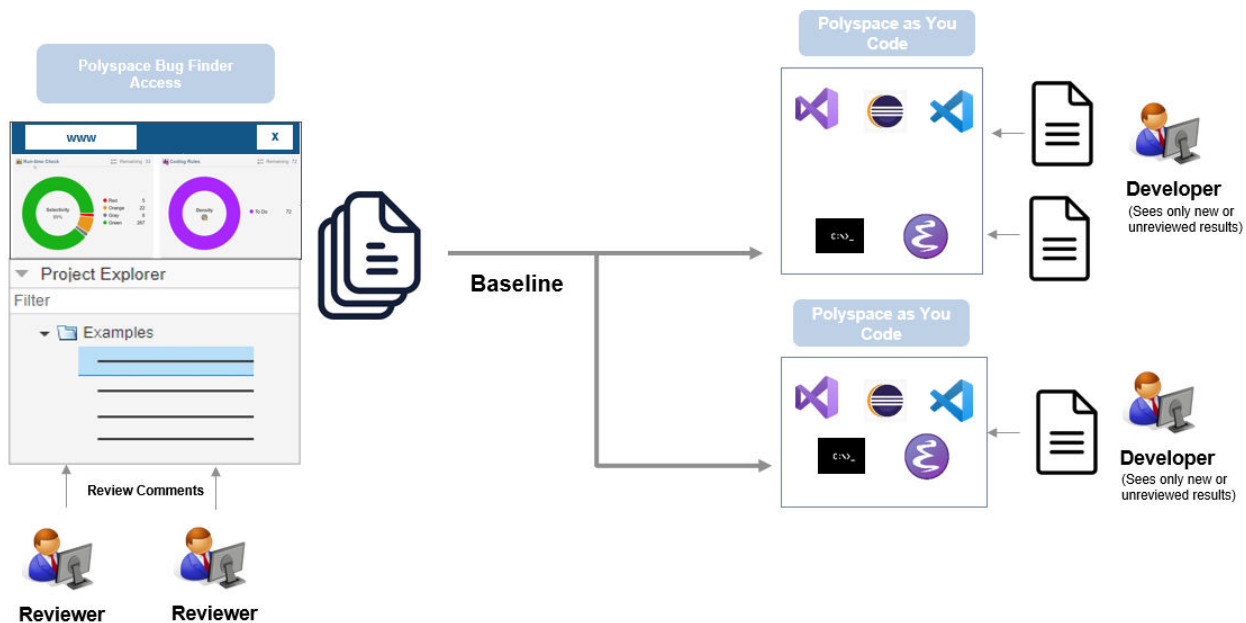
- “Configure Polyspace as You Code Extension in Visual Studio” on page 11-2
- “Run Polyspace as You Code in Visual Studio Code and Review Results” on page 29-6

Baseline Polyspace as You Code Results in Eclipse

This topic describes how to configure the Polyspace as You Code plugin in Eclipse. For Polyspace desktop products such as Polyspace Bug Finder, see “Bug Finder Analysis Based on Eclipse Projects”.

For more efficient bug fixing, you can baseline the results of a Polyspace as You Code analysis with previous results. When you baseline the results, you compare them against the results of a previous run and focus on new results only or on unreviewed results only. You baseline Polyspace as You Code results using previous Polyspace Bug Finder Server results that you download from Polyspace Access.

Note To keep using the most up-to-date baseline information, make sure that you periodically run the command to download results from Polyspace Access. See “Step 3: Download Baseline” on page 11-55.



What Baselined Results Look Like

If you baseline Polyspace as You Code results using Polyspace Bug Finder Server results that you downloaded from Polyspace Access, you can see the following benefits:

- Results that have a justified **Status** on Polyspace Access (**No Action Planned**, **Justified**, or **Not a Defect**) are no longer shown.
- Results that have a non-justified **Status** on Polyspace Access carry over all review information to the **Polyspace Results List** pane in Eclipse. If a result is reviewed in Polyspace Access and marked as such, *one of the following* is true:
 - The **Status** is different from **Unreviewed**.
 - The **Severity** is different from **Unset**.

- The **Comment** is not blank.
- If you specify the Polyspace as You Code plugin setting **Show only new findings compared to the results baseline**, you see only results that are new in the current run. That way, you can only focus on results that explicitly occurred because of the changes you made since the last Polyspace Server run.

Baselining Steps

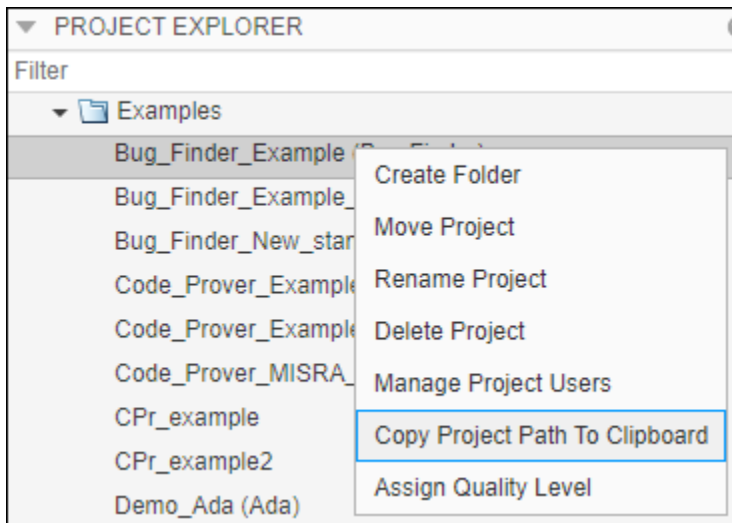
To use Polyspace Bug Finder Server results as baseline for a Polyspace as You Code analysis, follow the steps below. Once a baseline is downloaded, if you choose to point to the baseline, each subsequent run, whether on file save or on-demand, uses the baseline.

Step 1: Identify Project to Use as Baseline

First, identify a project in Polyspace Access that you want to use as baseline. The project must contain results of a Polyspace Bug Finder Server analysis on files that you will analyze in Polyspace as You Code.

Copy the path to the project for use in the Visual Studio Code extension settings. To copy this path:

- 1 Open the Polyspace Access web interface in a web browser.
- 2 On the **Project Explorer** pane, right-click the project and select **Copy Project Path to Clipboard**.



Step 2: Refer to Project from Polyspace as You Code


Next, refer to the Polyspace Access project from the Polyspace as You Code plugin settings in Eclipse.

- 1 Go to **Window > Preferences** and select the **Polyspace as You Code** node.
- 2 On the **Baseline** node configure these settings:
 - **Show Baseline Information:** Select an option from the dropdown.
 - Show local findings only — Do not use the baseline.

- **Show local findings and baseline info** — After you download a baseline, findings that are already justified in the baseline run are suppressed in the **Polyspace Problems** view.
- **Show new findings only** — After you download a baseline, findings that are already present in the baseline run are suppressed in the **Polyspace Problems** view. Use this option to focus on only new findings in your file.
- **Polyspace Access URL**: The URL of the server that hosts Polyspace Access. For instance, `https://my-access-url:9443/`.
- **Polyspace Access Login**: The username that you use to log in to Polyspace Access.
- **Project path**: The path to the project in Polyspace Access that you want to use as baseline. You previously copied this path from the Polyspace Access web interface.

Step 3: Download Baseline

Explicitly download the Polyspace Access result to use as baseline.

- 1 The first time you configure the plugin settings, click  in the **Baseline** view of the **Polyspace as You Code** perspective.

The plugin prompts you to enter your Polyspace Access password and you have the option to save the password so that you don't need to enter it in subsequent downloads.

Click the same icon to download an updated baseline. The icon is available only if you configure the plugin to use a baseline.

- 2 The plugin prompts you to enter your Polyspace Access password. You have the option to save the password so that you don't need to enter it in subsequent downloads.

An indicator in the **Baseline** view shows the progress of the download. If there are issues with the download, check the **Console** output.

After download, subsequent runs use the baseline. To disable baseline usage, set the **Show Baseline information** setting to **Show local findings only**.

See Also

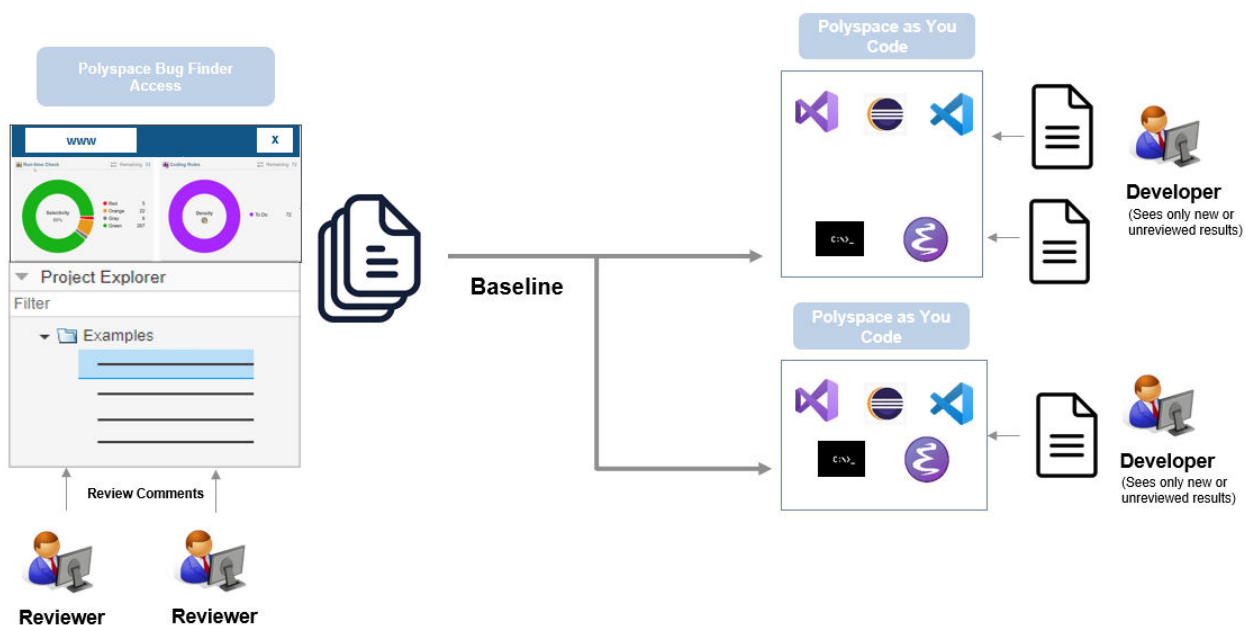
More About

- “Configure Polyspace as You Code Plugin in Eclipse” on page 11-19
- “Run Polyspace as You Code in Eclipse and Review Results” on page 29-15

Baseline Polyspace as You Code Results on Command Line

For more efficient bug fixing, you can baseline the results of a Polyspace as You Code analysis with previous results. When you baseline the results, you compare them against the results of a previous run and focus on new results only or on unreviewed results only. You baseline Polyspace as You Code results using previous Polyspace Bug Finder Server results that you download from Polyspace Access.

Note To keep using the most up-to-date baseline information, make sure that you periodically run the command to download results from Polyspace Access. See “Step 2: Download Baseline” on page 11-58.



What Baselined Results Look Like

The effect of baselining depends on whether you export results to the console or JSON format (SARIF). For more details on the formats, see `polyspace-results-export`.

Console Output

Results that have a justified **Status** on Polyspace Access (**No Action Planned**, **Justified**, or **Not a Defect**) are no longer shown in the console output.

JSON Output

In the following statements, *obj* represents the JSON object that is exported from the Polyspace results.

- If a result is new and not already present in Polyspace Access, the corresponding property `obj.runs[0].results[n].baselineState` is set to "new":

```
"baselineState" : "new"
```

Otherwise, the property is set to "unchanged".

- Results carry over their review information (**Status**, **Severity** and additional notes) from Polyspace Access to the corresponding properties in `obj.runs[0].results[n].properties`.

For instance, without a baseline, these properties are:

```
"severity" : "Unset",
"status" : "Unreviewed",
"comment" : ""
```

With a baseline, the `severity` can be different from "Unset", the `status` different from "Unreviewed", and so on.

- Results that have a justified **Status** on Polyspace Access (**No Action Planned**, **Justified**, or **Not a Defect**) appear with the property `obj.runs[0].results[n].properties.justified` set to `true`:

```
"justified" : true
```

Baselining Steps

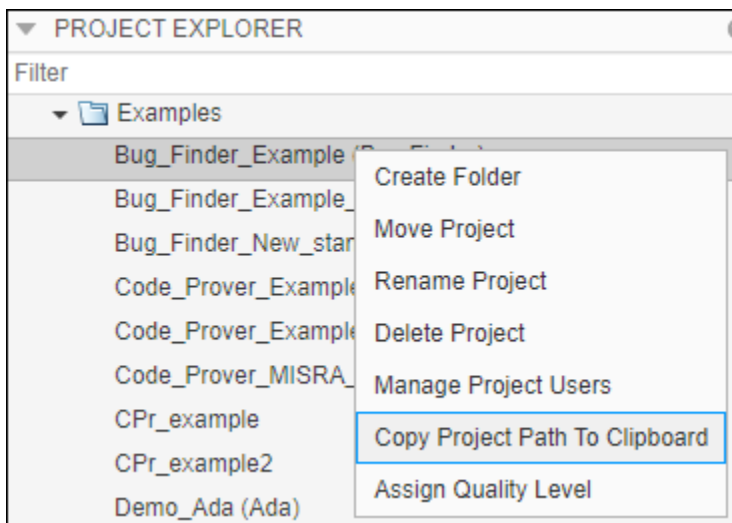
To use Polyspace Bug Finder Server results as baseline for a Polyspace as You Code analysis, follow the steps below.

Step 1: Identify Project to Use as Baseline

First, identify a project in Polyspace Access that you want to use as baseline. The project must contain results of a Polyspace Bug Finder Server analysis on files that you will analyze in Polyspace as You Code.

Copy the path to the project that you want to use as baseline. To copy this path:

- 1 Open the Polyspace Access web interface in a web browser.
- 2 On the **Project Explorer** pane, right-click the project and select **Copy Project Path to Clipboard**.



Step 2: Download Baseline

Next, download the baseline information from the Polyspace Access project. For instance, in a terminal, enter the following:

```
polyspace-access -host hostname -download projectPath -output-folder-path downloadFolder
```

Here:

- *hostname* is the name of the Polyspace Access server.
- *projectPath* is the path to the project on Polyspace Access that is used as baseline. You copied this name from the Polyspace Access web interface.
- *downloadFolder* is the folder to which you download the baseline information.

After download, the folder contains three databases: results (*ps_results.psb*), source files (*ps_sources.db*), and review information (*ps_comments.db*). You cannot open these results in the Polyspace user interface or use them in any other way other than as baseline for Polyspace as You Code runs.

The folder also contains a file *ps_access_info.json* that contains information about the Access project and run ID that was used as baseline. If required, you can write a script to compare this run ID with the latest run ID of the project on Polyspace Access and run this script at certain points in your workflow to make sure that you are always using the latest run of the project as baseline.

Step 3: Use Baseline

Once the baseline download completes, import the downloaded baseline information into your current analysis by using option `-import-comments`. In a terminal, enter the following:

```
#Linux command
polyspace-bug-finder-access -sources filename -import-comments downloadFolder \
-results-dir resultsFolder

#DOS command
polyspace-bug-finder-access.exe -sources filename -import-comments downloadFolder ^
-results-dir resultsFolder
```

Here:

- *filename* is the current file being analyzed.
- *downloadFolder* is the folder to which you previously downloaded the baseline information.
- *resultsFolder* is the folder for storing analysis results.

You can see the effects of using the baseline when you export the analysis results by using the `polyspace-results-export` command. For instance:

```
polyspace-results-export -format console -results-dir resultsFolder
```

.See also “What Baselined Results Look Like” on page 11-56.

See Also

`polyspace-bug-finder-access` | `polyspace-results-export`

More About

- “Run Polyspace as You Code from Command Line and Export Results” on page 29-21

Configure Checkers for Polyspace as You Code in Eclipse

In this section...

“Select Checkers and Coding Rules” on page 11-60

“Modify Checker Behavior” on page 11-62

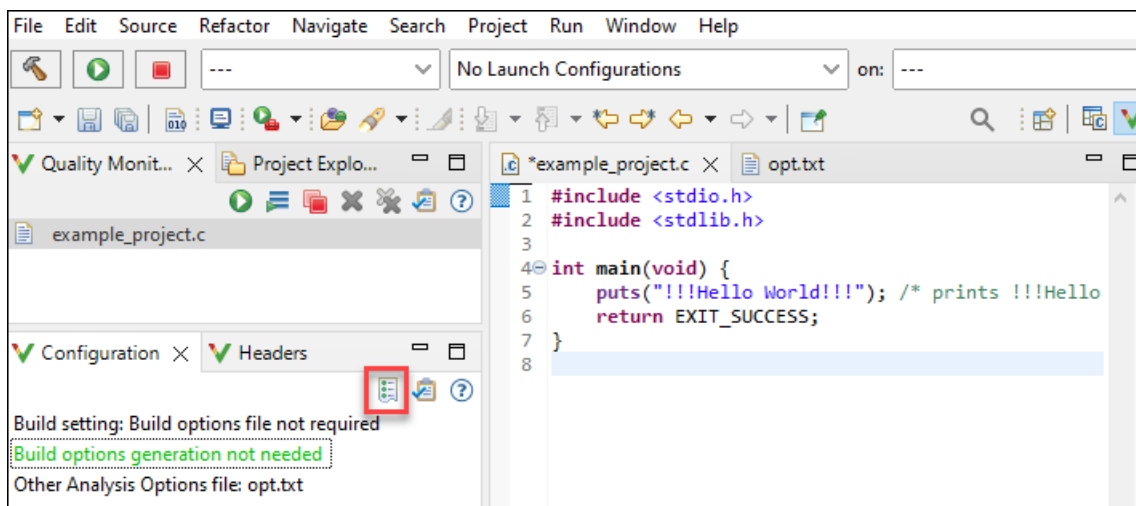
This topic describes how to configure the Polyspace as You Code plugin in Eclipse. For Polyspace desktop products such as Polyspace Bug Finder, see “Bug Finder Analysis Based on Eclipse Projects”.

You can check for various types of defects and coding rule violations by using Polyspace as You Code in Eclipse. See Defects and Coding Standards. The default analysis checks for a subset of defects. See “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 18-65. To check for nondefault defects and coding rule violations, configure Polyspace as You Code extension in your IDE.

To configure checkers, create a checkers file, and then specify the checkers file in the Configure Project window. For equivalent workflows in the Polyspace desktop and server, see “Prepare Checkers Configuration for Polyspace Bug Finder Analysis” on page 18-38.


Select Checkers and Coding Rules

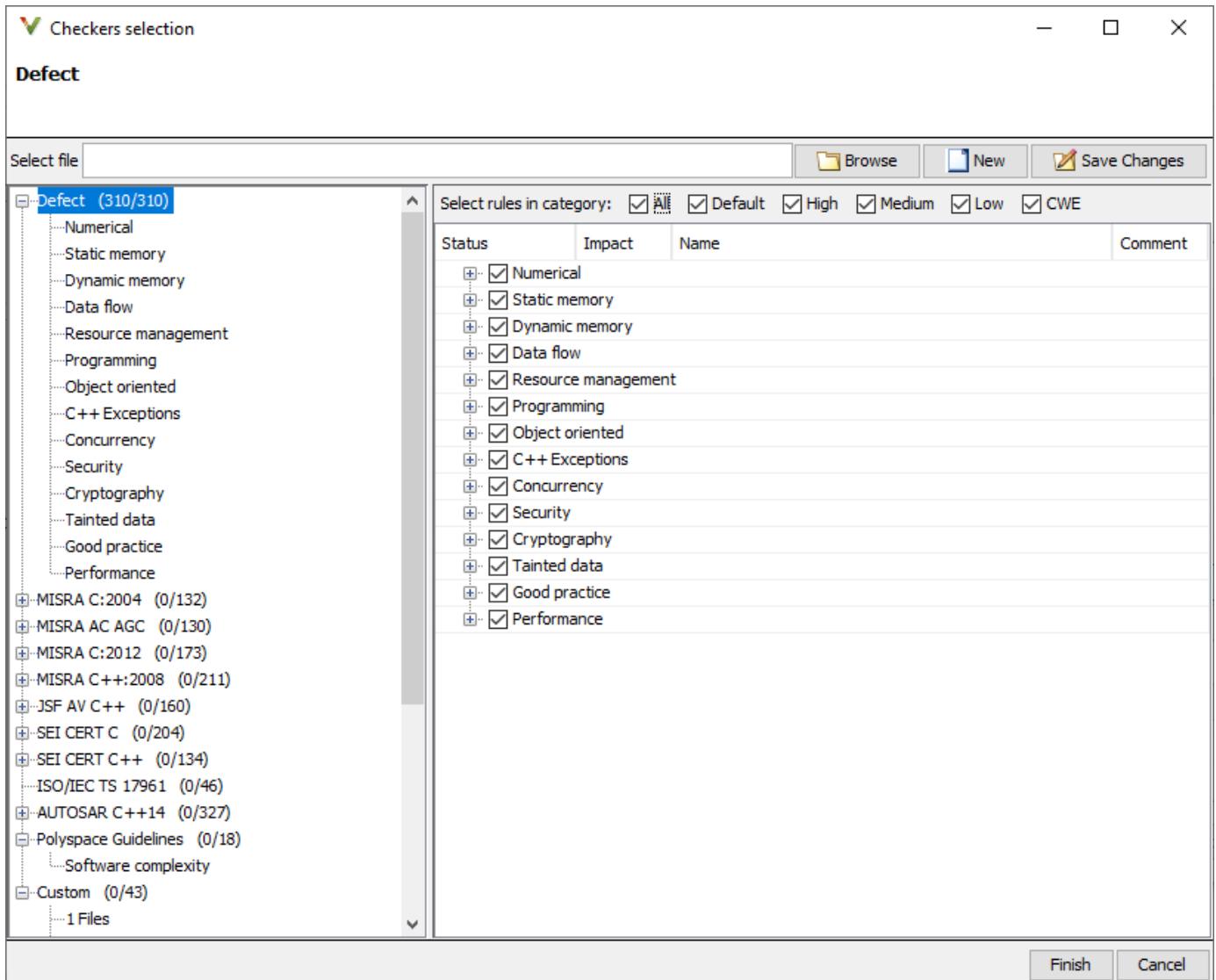
To select coding rule checkers and nondefault defect checkers, click the **Configure Checkers** icon in the **Configuration View** of the **Polyspace as You Code** perspective. Configure the checker selection in the Configure Project window.



Create or Modify Checkers Configuration

Create a new selection or modify an existing selection of checkers and coding rules in the Checker selection window. Save the new configuration in a reusable checkers file.

- 1 In the **Configuration** view of the **Polyspace as You Code** perspective, open the **Checkers selection** window by clicking  .



- 2 To create a new selection, in the Checkers Selection window, select the defect and the coding rule checkers that you want to activate. To modify an existing selection, click **Browse**, navigate to the existing checkers file and then modify the checkers selection.

You can also select predefined categories of defect checkers such as **All**, **Default**, **High**, **Medium**, and **Low**. See “Classification of Defects by Impact” on page 18-49. Similarly, you can activate a predefined set of coding rules that are defined by their corresponding standards.

- When selecting **Guidelines** > **Software Complexity** checkers, review their thresholds. If the default thresholds are not acceptable, specify a suitable threshold in the **Threshold** column. See `Check guidelines (-guidelines)`.
- When selecting **Custom** rules, review the **Pattern** and **Convention** for the rules. See `Check custom rules (-custom-rules)`.
- To create a custom coding standard classification, or to tag coding rule checkers of interest, enter text in the **Comment** column of the **Checkers selection** window. Polyspace displays that text in the **Results Details** pane and in the **Detail** column of the **Results List** (if

available) when you review results in the desktop interface, in Polyspace Access, or in the Polyspace as You Code plugins.

- 3 Save the selection as a reusable checkers XML file by clicking **Save Changes**. After you click **Finish**, the path to the new checkers XML file is specified in the field **Checkers file** in the Configure Project window.

Import Checkers Configuration from Desktop Project

You can import checkers and coding rule configuration from a Polyspace desktop project (*.psprj) file. See “Import Analysis Options from Polyspace Desktop Project” on page 11-39.

Modify Checker Behavior

To modify the default behavior of Bug Finder defect checkers and coding rules, use analysis options. For a list of analysis options that modify the default checker behavior, see “Modify Default Behavior of Bug Finder Checkers” on page 18-3.

To specify analysis options in Polyspace as You Code:

- Append the analysis options in the options file specified in the field **Analysis options file**. An options file is a text file with one analysis option for each line. For instance, to add the analysis options `-code-behavior-specifications` and `Effective boolean types (-boolean-types)`, in the options file, append these lines:

```
-code-behavior-specifications file1  
-boolean-types boolean1_t,boolean2_t
```
- If you do not have an option file, create an option file that contains the necessary options. Specify the path to the new options file in the field **Analysis options file**. See “Options Files for Polyspace Analysis” on page 12-5.

See Also

More About

- “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 18-65
- “Options Files for Polyspace Analysis” on page 12-5
- “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78
- “Modify Default Behavior of Bug Finder Checkers” on page 18-3

Configure Checkers for Polyspace as You Code in Visual Studio

In this section...

“Select Checkers and Coding Rules” on page 11-63

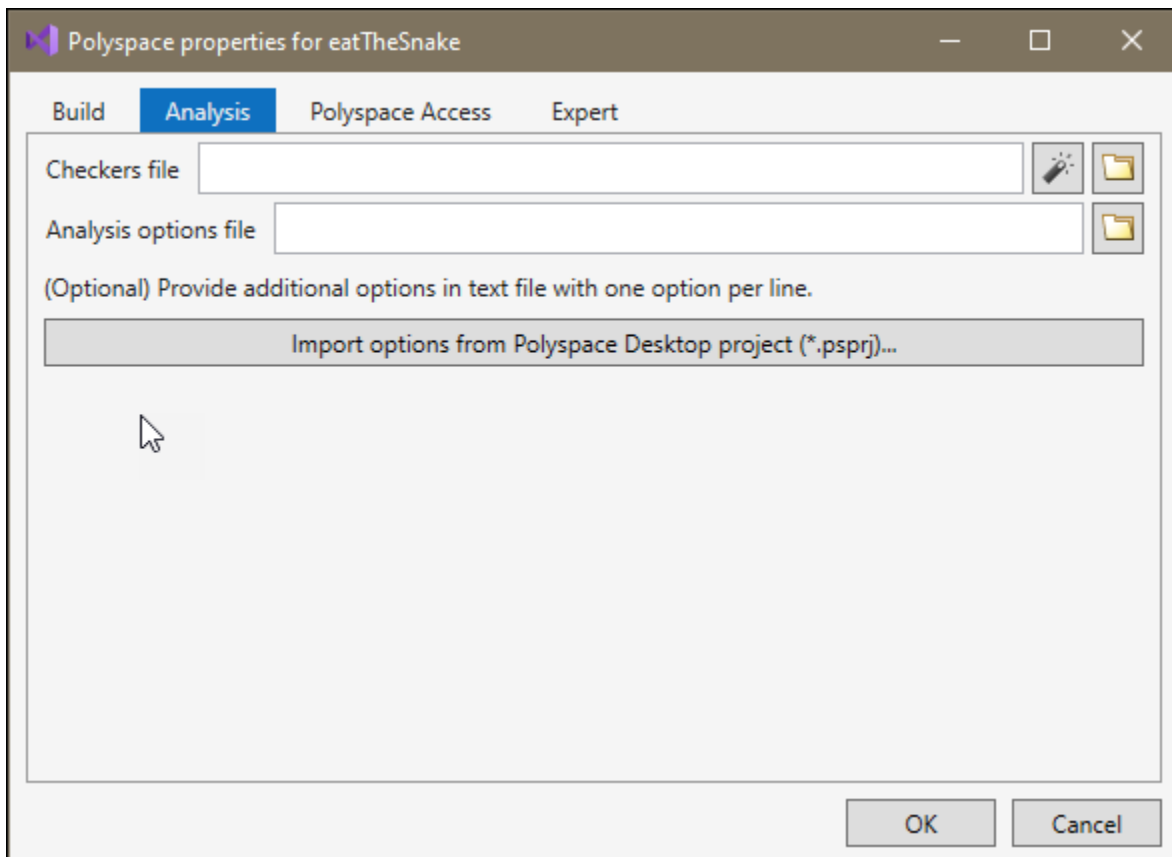
“Modify Checker Behavior” on page 11-65

You can check for various types of defects and coding rule violations by using Polyspace as You Code in Visual Studio. See Defects and Coding Standards. The default analysis checks for a subset of defects. See “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 18-65. To check for nondefault defects and coding rule violations, configure Polyspace as You Code extension in your IDE.

To configure checkers, create a checkers file, and then specify the checkers file in the Options window. For equivalent workflows in the Polyspace desktop and server, see “Prepare Checkers Configuration for Polyspace Bug Finder Analysis” on page 18-38.

Select Checkers and Coding Rules

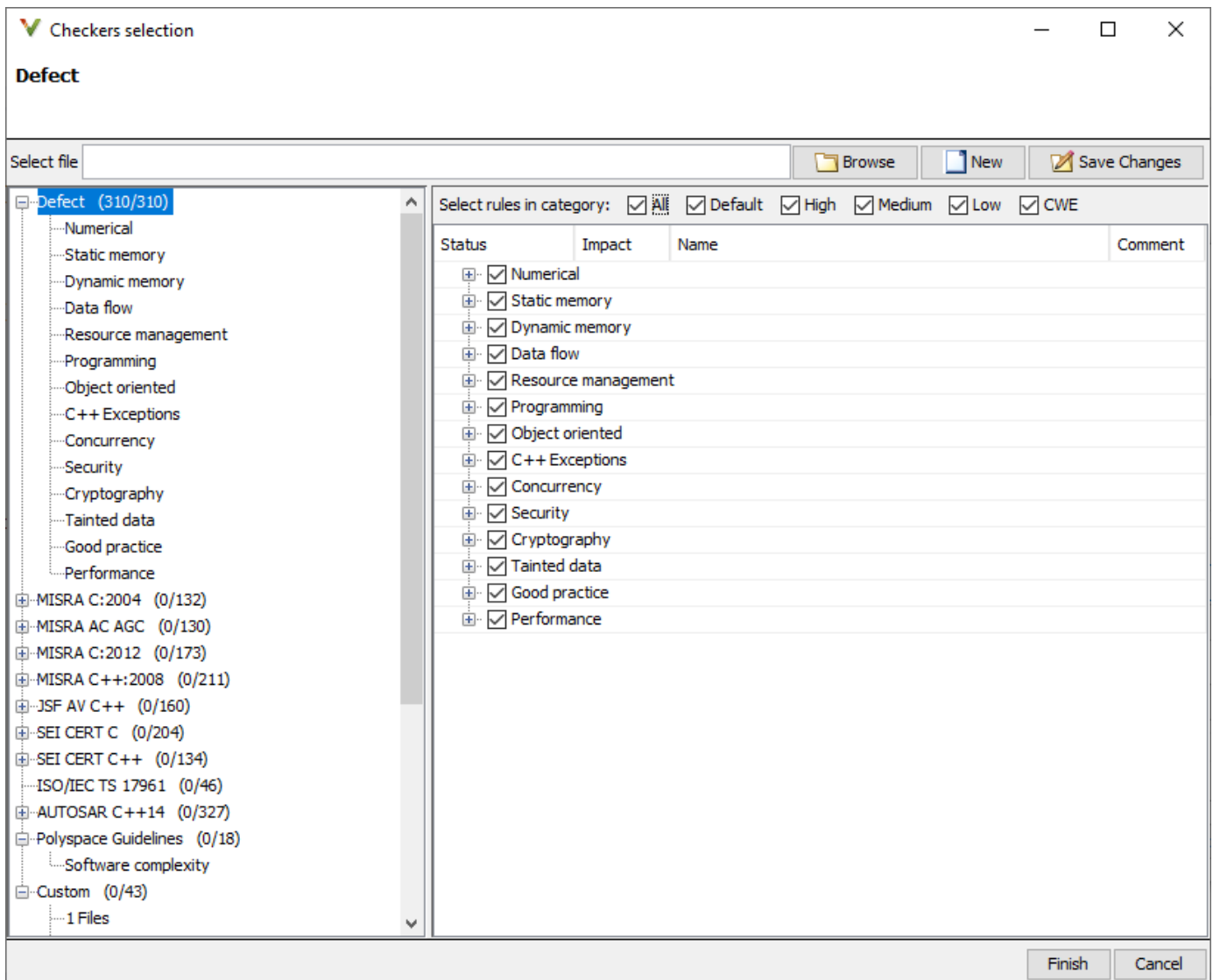
To select coding rule checkers and nondefault defect checkers, right-click the project on the **Solution Explorer** pane and select **Polyspace properties**. Specify the checkers configuration on the **Analysis** tab.



Create or Modify Checkers Configuration

Create a new selection or modify an existing selection of checkers and coding rules in the **Checker selection** window. Save the new configuration in a reusable checkers file.

- 1 On the **Analysis** tab in the **Polyspace properties** window, click  to open the **Checkers selection** window.



- 2 To create a new selection, in the **Checkers selection** window, select the defect and the coding rule checkers that you want to activate. To modify an existing selection, click **Browse**, navigate to the existing checkers file and then modify the checkers selection.

You can also activate predefined categories of defect checkers such as **All**, **Default**, **High**, **Medium**, and **Low**. See “Classification of Defects by Impact” on page 18-49. Similarly, you can activate predefined set of coding rules that are defined by their corresponding standards.

- When selecting **Guidelines > Software Complexity** checkers, review their thresholds. If the default thresholds are not acceptable, specify a suitable threshold in the **Threshold** column. See `Check guidelines (-guidelines)`.
 - When selecting **Custom** rules, review the **Pattern** and **Convention** for the rules. See `Check custom rules (-custom-rules)`.
 - To create a custom coding standard classification, or to tag coding rule checkers of interest, enter text in the **Comment** column of the **Checkers selection** window. Polyspace displays that text in the **Results Details** pane and in the **Detail** column of the **Results List** (if available) when you review results in the desktop interface, in Polyspace Access, or in the Polyspace as You Code plugins.
- 3 Save the selection as a reusable checkers XML file by clicking **Save Changes**. After you click **Finish**, the path to the new checkers XML file is specified in the field **Checkers file** on the **Analysis** tab in the **Polyspace properties** window.

Import Checkers Configuration from Desktop Project

To import checkers and coding rule selection from a Polyspace desktop project (*.psprj) file, on the **Analysis** tab in the **Polyspace properties** window, click **Import options from Polyspace desktop project (*.psprj)**. Browse to the folder containing the project file and specify the project file. The checkers configuration in the desktop project is extracted as a checkers file, which is specified in the field **Checkers file**. The analysis options of the desktop project are extracted as an options file which is specified in the field **Analysis options file**.

Modify Checker Behavior

To modify the default behavior of Bug Finder defect checkers and coding rules, use analysis options. For a list of analysis options that modify the default checker behavior, see “Modify Default Behavior of Bug Finder Checkers” on page 18-3.

To specify analysis options in Polyspace as You Code:

- Append the analysis options in the options file specified in the field **Analysis options file**. An options file is a text file with one analysis option for each line. For instance, to add the analysis options `-code-behavior-specifications` and `Effective boolean types (-boolean-types)`, in the options file, append these lines:


```
-code-behavior-specifications file1
-boolean-types boolean1_t,boolean2_t
```
- If you do not have an option file, create an option file that contains the necessary options. Specify the path to the new options file in the field **Analysis options file**. See “Options Files for Polyspace Analysis” on page 12-5.

See Also

More About

- “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 18-65
- “Options Files for Polyspace Analysis” on page 12-5
- “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78
- “Modify Default Behavior of Bug Finder Checkers” on page 18-3

Configure Checkers for Polyspace as You Code in Visual Studio Code

In this section...
“Configure Checkers in Checkers File” on page 11-66
“Modify Checkers Behavior” on page 11-68

You can check for various types of defects and coding rule violations by using Polyspace as You Code in Visual Studio Code. See Defects and Coding Standards. The default analysis checks for a subset of defects. See “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 18-65. To check for nondefault defects and coding rule violations, configure Polyspace as You Code extension in your IDE.

To configure checkers, create a checkers file, and then specify the checkers file in extension settings. For equivalent workflows in the Polyspace desktop and server, see “Prepare Checkers Configuration for Polyspace Bug Finder Analysis” on page 18-38.

Configure Checkers in Checkers File

To configure checkers, first select checkers in a checkers file. Then specify the checkers file in the **Settings** pane.

Step 1: Select Checkers and Coding Rules

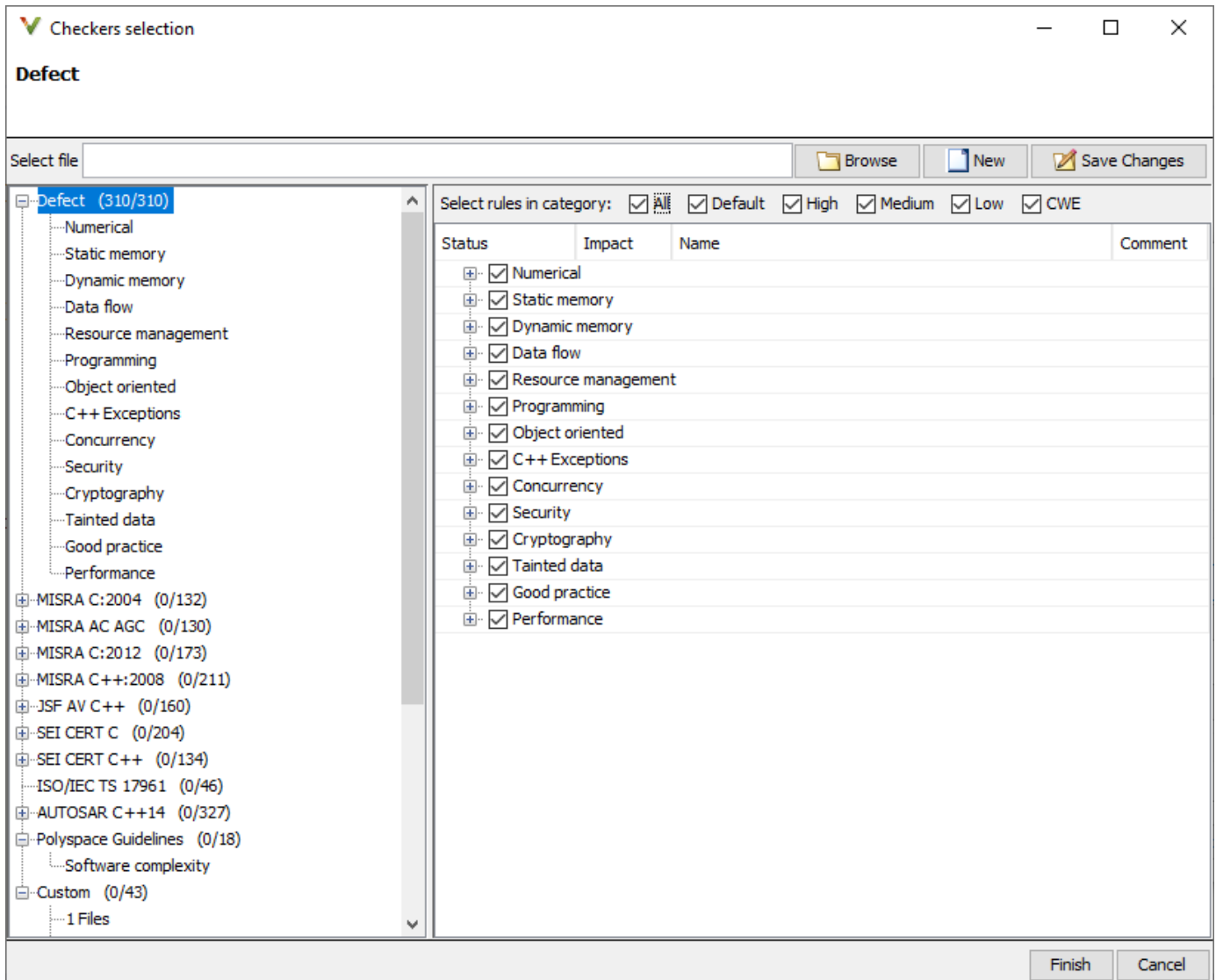
To enable nondefault defects and coding rules, you might:

- Create or modify a checkers file.
- Import a checkers selection from a Polyspace desktop project.

Create or Modify Checkers File

Create a new selection or modify an existing selection of checkers and coding rules in the Checker selection window. Save the new configuration in a reusable checkers file. To use an existing checkers file without modifying the checkers selection, specify a path to it in the **Settings** pane. See “Step 2: Specify Checkers File in Extension Settings” on page 11-68.

- 1 To open the **Checkers selection** user interface, in the command palette, run **Polyspace: Configure Checkers**.



- 2 To create a new selection, in the Checkers Selection window, select the defect and the coding rule checkers that you want to activate. To modify an existing selection, click **Browse**, navigate to the existing checkers file and then modify the checkers selection.

You can also activate predefined categories of defect checkers such as **All**, **Default**, **High**, **Medium**, and **Low**. See “Classification of Defects by Impact” on page 18-49. Similarly, you can activate predefined set of coding rules that are defined by their corresponding standards.

- When selecting **Guidelines > Software Complexity** checkers, review their thresholds. If the default thresholds are not acceptable, specify a suitable threshold in the **Threshold** column. See *Check guidelines (-guidelines)*.
- When selecting **Custom** rules, review the **Pattern** and **Convention** for the rules. See *Check custom rules (-custom-rules)*.
- To create a custom coding standard classification, or to tag coding rule checkers of interest, enter text in the **Comment** column of the **Checkers selection** window. Polyspace displays that text in the **Results Details** pane and in the **Detail** column of the **Results List** (if

available) when you review results in the desktop interface, in Polyspace Access, or in the Polyspace as You Code plugins.

- 3 Save the selection as a reusable checkers XML file by clicking **Save Changes**. You can later reuse the checkers XML file as an input in the field **Checkers file**. Click **Finish**.

Import Checkers Configuration from desktop Project

If you have a Polyspace desktop project file (*.psprj), you can import checkers configuration from it. In the Visual Studio Code terminal, run:

```
#Linux command
polyspace-checkers-selection -checkers-selection-output-file PathToOutputFile.json \
-import-options-from-psprj PathToProject.psprj

#DOS command

polyspace-checkers-selection.exe -checkers-selection-output-file PathToOutputFile.json ^
-import-options-from-psprj PathToProject.psprj
```


where *PathToProject.psprj* is the full path to the polyspace desktop project file and *PathToOutputFile.json* is the full path to a JSON file. The JSON file must be in a writable folder. The JSON file contains the location of the produced checkers file in this format:

```
{
  "checkers-activation-file": "GeneratedCheckersActivationFile",
  "analysis-options-file": "GeneratedAnalysisOptionFile"
}
```

The checkers file in *GeneratedCheckersActivationFile* contains the imported checker configurations from the Polyspace desktop project file.

Step 2: Specify Checkers File in Extension Settings

After creating the checkers file, specify the path to it in the **Settings** pane:

- On the Side bar, click the **Extensions** button. The **Extensions** pane opens where your installed extensions are listed.
- Locate Polyspace as You Code in the **Extensions** pane. Click  and select **Extension Settings**.
- In the **Settings** pane, specify the path to the checkers file in the field **Checkers File**.

Modify Checkers Behavior

To modify the default behavior of Bug Finder defect checkers and coding rules, use analysis options. For a list of analysis options that modify the default checker behavior, see “Modify Default Behavior of Bug Finder Checkers” on page 18-3.

To specify analysis options in Polyspace as You Code:

- Append the analysis options in the options file specified in the field **Analysis options file**. An options file is a text file with one analysis option for each line. For instance, to add the analysis options `-code-behavior-specifications` and Effective boolean types (`-boolean-types`), in the options file, append these lines:

```
-code-behavior-specifications file1
-boolean-types boolean1_t,boolean2_t
```

- If you do not have an option file, create an option file that contains the necessary options. Specify the path to the new options file in the field **Other Analysis Options**. See “Options Files for Polyspace Analysis” on page 12-5.

See Also

More About

- “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 18-65
- “Options Files for Polyspace Analysis” on page 12-5
- “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78
- “Modify Default Behavior of Bug Finder Checkers” on page 18-3

Configure Checkers for Polyspace as You Code at the Command Line

In this section...

“Configure Checkers and Coding Rules Directly at the Command Line” on page 11-70

“Configure Checkers in Checkers file” on page 11-71

“Modify Checkers Behavior” on page 11-73

If you use an unsupported IDE, you can check for various types of defects and coding rule violations by using Polyspace as You Code at the command line. See Defects and Coding Standards. The default analysis checks for a subset of defects. See “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 18-65. To check for other defects and coding rule violations, configure Polyspace as You Code.

To configure checkers, create a checkers file and then specify the checkers file at the command line. For equivalent workflows in the Polyspace desktop and server, see “Prepare Checkers Configuration for Polyspace Bug Finder Analysis” on page 18-38.

Configure Checkers and Coding Rules Directly at the Command Line

When running Polyspace as You Code in an unsupported IDE, you can specify a selection of checkers and coding rules by using these analysis options with appropriate values directly at the command line:

- Find defects (-checkers)
- Check MISRA C:2004 (-misra2)
- Check MISRA C:2012 (-misra3)
- Check SEI CERT-C (-cert-c)
- Check ISO/IEC TS 17961 (-iso-17961)
- Check MISRA C++:2008 (-misra-cpp)
- Check SEI CERT-C++ (-cert-cpp)
- Check AUTOSAR C++ 14 (-autosar-cpp14)
- Check JSF AV C++ rules (-jsf-coding-rules)
- Check custom rules (-custom-rules)
- Check guidelines (-guidelines)
- Check CWE (-cwe)

For instance, to activate the performance checkers and MISRA C:2012 coding rule, in the command line interface, run

```
polyspace-bug-finder-access -sources <source.c> -checkers performance -misra3 all
```

See the documentation of the analysis options for their command line syntax. To view the results, use `polyspace-results-export`.

Specifying checkers and coding rule selection enables you to select predefined subsets of checkers and coding rules. To select a customized subset of checkers and coding rules, configure checkers by using a checker file.

Configure Checkers in Checkers file

To configure checkers, first select checkers in a checkers file. Then specify the checkers file in the **Settings** pane.

Step 1: Select Checkers and Coding Rule

To enable nondefault defects and coding rules, you might:

- Create or modify a checkers file.
- Import a checkers selection from a Polyspace desktop project.

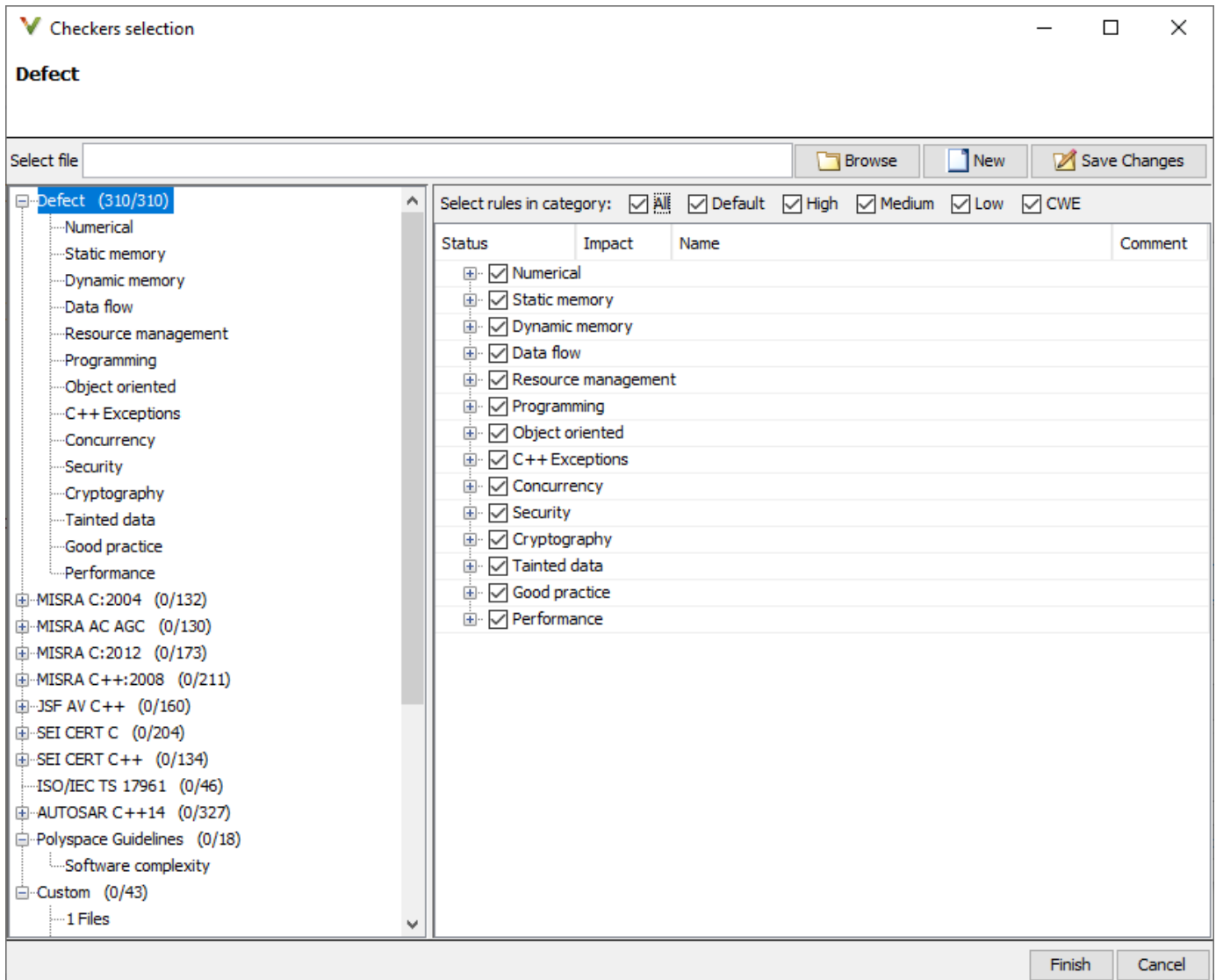
Create or Modify Checkers Configuration

Create a new selection or modify an existing selection of checkers and coding rules in the Checker selection window. Save the new configuration in a reusable checkers file. To use an existing checkers file without modifying the checkers selection, specify it at the command line. See “Step 2: Specify Checker File at the Command Line” on page 11-73.

- 1 To open the Checkers selection user interface, in the command line, run:

```
polyspace-checkers-selection
```

The Checkers Selection interface opens.



- 2 To create a new selection, in the Checkers Selection window, select the defect and the coding rule checkers that you want to activate. To modify an existing selection, click **Browse**, navigate to the existing checkers file and then make your selection.

You can also activate predefined categories of defect checkers such as **All**, **Default**, **High**, **Medium**, and **Low**. See “Classification of Defects by Impact” on page 18-49. Similarly, you can activate predefined set of coding rules that are defined by their corresponding standards.

- When selecting **Guidelines** > **Software Complexity** checkers, review their thresholds. If the default thresholds are not acceptable, specify a suitable threshold in the **Threshold** column. See Check guidelines (-guidelines).
- When selecting **Custom** rules, review the **Pattern** and **Convention** for the rules. See Check custom rules (-custom-rules).
- To create a custom coding standard classification, or to tag coding rule checkers of interest, enter text in the **Comment** column of the **Checkers selection** window. Polyspace displays that text in the **Results Details** pane and in the **Detail** column of the **Results List** (if

available) when you review results in the desktop interface, in Polyspace Access, or in the Polyspace as You Code plugins.

- 3 Save the selection as a reusable checkers XML file as *CreatedCheckerFile.xml* and then **Finish**. You can later reuse *CreatedCheckerFile.xml* as a value to `-checkers-activation-file`.

Import Checkers Configuration from Desktop Project

If you have a Polyspace desktop Project (*.psprj) file, you can import checker selection from it. At the command line, run

```
#Linux command
polyspace-checkers-selection -checkers-selection-output-file PathToOutputFile.json \
-import-options-from-psprj PathToProject.psprj
```

```
#DOS command
polyspace-checkers-selection -checkers-selection-output-file PathToOutputFile.json ^
-import-options-from-psprj PathToProject.psprj
```

where *PathToProject.psprj* is the full path to the polyspace desktop project file and *PathToOutputFile.json* is the full path to a JSON file. The JSON file must be in a writable folder. The JSON file contains the location of the produced checkers file in this format:

```
{
  "checkers-activation-file": "PathToCreatedCheckerFile",
  "analysis-options-file": "CreatedOptionsFile"
}
```

The checkers file in *PathToCreatedCheckerFile* contains the checker configurations in the Polyspace desktop project file.

Step 2: Specify Checker File at the Command Line

After you obtain the checkers file, specify its full path as an argument to `-checkers-activation-file`. For instance:

```
#Linux command
polyspace-bug-finder-access -sources <source.c> \
-checkers-activation-file PathToCreatedCheckerFile
```

```
#DOS command
```

```
polyspace-bug-finder-access -sources <source.c> ^
-checkers-activation-file PathToCreatedCheckerFile
```

where *PathToCreatedCheckerFile* is the full path to the checkers file.

Modify Checkers Behavior

To modify the default behavior of Bug Finder defect checkers and coding rules, use analysis options. For a list of analysis options that modify the default checker behavior, see “Modify Default Behavior of Bug Finder Checkers” on page 18-3.

To specify analysis options in Polyspace as You Code:

- Use the options in the command line. For instance, to modify the trust boundary of your analysis, in the command line, run:

```
#Linux Command
polyspace-bug-finder-access -sources <source.c> -checkers-activation-file CreatedCheckerFile\
```

```
-consider-analysis-perimeter-as-trust-boundary
#DOS command
polyspace-bug-finder-access -sources <source.c> -checkers-activation-file CreatedCheckerFile^
-consider-analysis-perimeter-as-trust-boundary
```

You can specify multiple behavior modifying options in the command line.

- Append the analysis options in the options file specified in the field **Analysis options file**. An options file is a text file with one analysis option for each line. For instance, to add the analysis options `-code-behavior-specifications` and Effective boolean types (`-boolean-types`), in the options file, append these lines:

```
-code-behavior-specifications file1
-boolean-types boolean1_t,boolean2_t
```

- If you do not have an existing options file, create an options file containing the necessary options. See “Options Files for Polyspace Analysis” on page 12-5.

See Also

More About

- “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 18-65
- “Options Files for Polyspace Analysis” on page 12-5
- “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78
- “Modify Default Behavior of Bug Finder Checkers” on page 18-3
- “Run Polyspace as You Code from Command Line and Export Results” on page 29-21

Analysis Scope of Polyspace as You Code

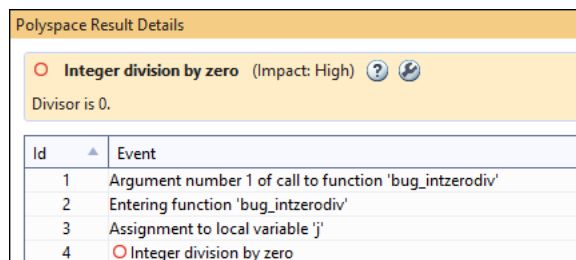
Polyspace as You Code is a static code analysis software meant for regular use by C/C++ developers within their Integrated Development Environments (IDEs). Polyspace as You Code can find bugs and coding standard violations on the file that is currently active in the IDE.

This topic outlines the analysis scope of Polyspace as You Code and the benefits of using Polyspace Bug Finder or Polyspace Bug Finder Server for full integration analysis.

Results Involve Current File Only

Polyspace as You Code is designed to provide results that are of immediate interest to developers. So the tool only shows results in files that you are currently working on. After installing the Polyspace as You Code extension, each time you open or save a file in your IDE, the analysis runs silently in the background and highlights issues in the file.

All issues found originate within the source file itself and can also be fixed within this file. You can either implement the fix at the highlighted location or another related location still within the current file. For instance, the following integer division by zero result is shown with related events on previous lines. You can implement a guard against division by zero just before the division or implement some checks on inputs to the function where the division is performed.



The screenshot shows a 'Polyspace Result Details' window. At the top, there is a red circle icon followed by the text 'Integer division by zero (Impact: High)'. Below this, it says 'Divisor is 0.'. A table below lists four events:

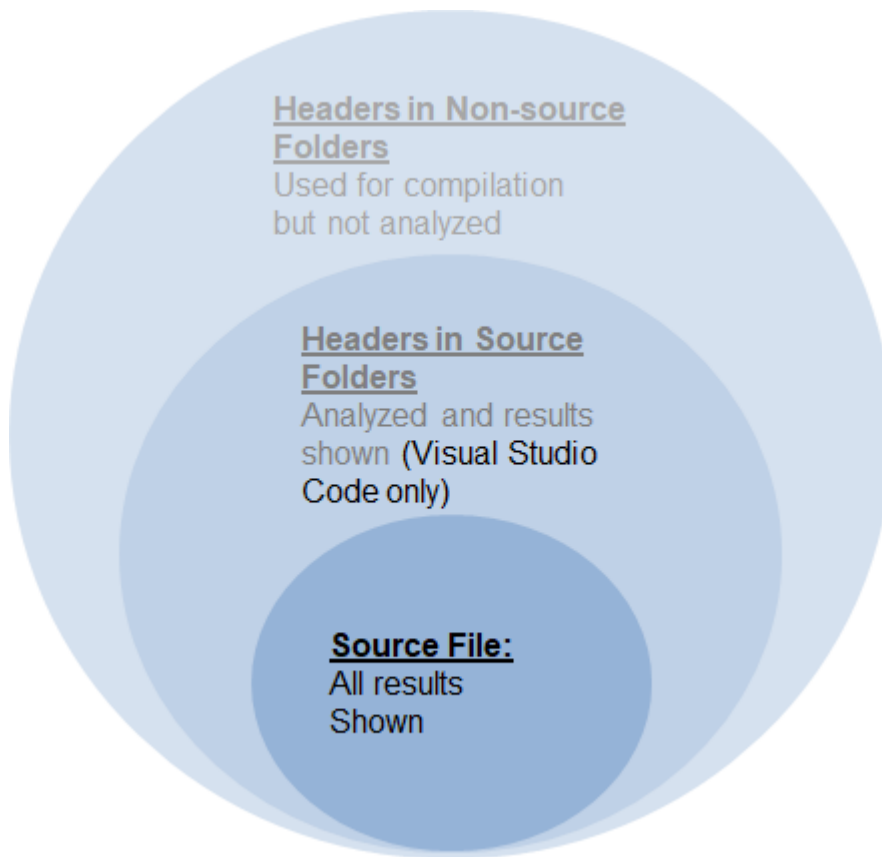
Id	Event
1	Argument number 1 of call to function 'bug_intzerodiv'
2	Entering function 'bug_intzerodiv'
3	Assignment to local variable 'j'
4	Integer division by zero

Results that involve multiple files, for instance, declaration mismatch across files or data flow between functions in different files, are not shown in the default Polyspace as You Code analysis. To see complete integration results on your project, analyze your project with Polyspace Bug Finder on your desktop or with Polyspace Bug Finder Server on a continuous integration (CI) server.

Some checkers that are not likely to find issues in a single-file analysis are completely disabled in Polyspace as You Code. See “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78.

Headers Included in Current File Not Analyzed

When you analyze a source file that `#include-s` a headers, that are in the same folder as the source (or in subfolders), Polyspace also analyzes these headers and in the case of the Polyspace as You Code extension in Visual Studio code, reports findings if it finds any. You can see which headers contain findings in the **HEADERS** view of the extension. See “Open Additional Polyspace Views” on page 29-7. All other header files are taken into account for compilation but not analyzed further.



The reason for this default behavior is the following:

- *Headers close to sources:*

The underlying assumption is that headers in the source folders are more closely related to the current source file and are therefore relevant for the analysis. A developer might want to fix issues reported in these headers.

- *Headers in non-source folders:*

Headers in other folders typically come from third party libraries and are not analyzed.

You can change this default behavior using these options:

- **Generate results for sources and (-generate-results-for):** Use this option to expand the scope of which headers must be analyzed.
- **Do not generate results for (-do-not-generate-results-for):** Use this option to expand the scope of which headers must not be analyzed.

You can also see findings in header files by explicitly analyzing the headers.

See Also

More About

- “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78

Checkers Deactivated in Polyspace as You Code Analysis

Polyspace as You Code finds bugs and coding rule violations in the currently active file in your IDE. If finding an issue requires multiple source files, the analysis cannot flag the issue by design. To help you with the checkers selection strategy:

- Checkers that typically produce most results in a multi-file context are completely deactivated in Polyspace as You Code.
- Some coding standard checkers cover both single and multi-file issues. These checkers are not deactivated but have a reduced scope and show fewer issues in Polyspace as You Code compared to a full integration analysis.

You can detect these multi-file issues by running an integration analysis on your project by using Polyspace Bug Finder or Polyspace Bug Finder Server.

Checkers and Coding Rule Deactivated in Polyspace as You Code

Checkers corresponding to these issues are deactivated in a Polyspace as You Code analysis. Even if you enable these checkers in your Polyspace as You Code IDE extension settings (or using the option `-checkers-activation-file`), they are internally deactivated during analysis.

Deactivated Bug Finder Checkers

The Bug Finder checkers that are deactivated in the default Polyspace as You Code analysis include:

- Declaration mismatch
- Qualifier removed in conversion
- Typedef mismatch
- “Concurrency Defects”

Deactivated CERT C Rules

The CERT C coding rules that are deactivated in the default Polyspace as You Code analysis include:

- CERT C: Rule DCL40-C
- CERT C: Rec. DCL15-C

Deactivated Cert C++ Rules

The CERT C++ coding rules that are deactivated in the default Polyspace as You Code analysis include:

- CERT C++: DCL40-C

Deactivated MISRA C:2004 and MISRA AC AGC Rules

The MISRA C:2004 and MISRA AC AGC coding rules that are deactivated in the default Polyspace as You Code analysis include:

- MISRA C:2004 and MISRA AC AGC Rules 5.1, 5.4, 5.6, 8.4, 8.9, 8.10. See “Supported MISRA C:2004 and MISRA AC AGC Rules” on page 17-9

Deactivated MISRA C:2012 Rules

The MISRA C:2012 coding rules that are deactivated in the default Polyspace as You Code analysis include:

- MISRA C:2012 Rule 2.3
- MISRA C:2012 Rule 2.4
- MISRA C:2012 Rule 2.5
- MISRA C:2012 Rule 5.1
- MISRA C:2012 Rule 5.6
- MISRA C:2012 Rule 5.8
- MISRA C:2012 Rule 5.9
- MISRA C:2012 Rule 8.3
- MISRA C:2012 Rule 8.6
- MISRA C:2012 Rule 8.7

Deactivated ISO/IEC TS 17961 Rules

The ISO/IEC TS 17961 coding rules that are deactivated in the default Polyspace as You Code analysis include:

- ISO/IEC TS 17961 [funcdecl]

Deactivated MISRA C++:2008 Rules

The MISRA C++:2008 coding rules that are deactivated in the default Polyspace as You Code analysis include:

- MISRA C++:2008 Rule 0-1-3
- MISRA C++:2008 Rule 2-10-5
- MISRA C++:2008 Rule 3-2-1
- MISRA C++:2008 Rule 3-2-2
- MISRA C++:2008 Rule 3-2-4
- MISRA C++:2008 Rule 15-4-1

Deactivated AUTOSAR C++14 Rules

The AUTOSAR C++14 coding rules that are deactivated in the default Polyspace as You Code analysis include:

- AUTOSAR C++14 Rule M0-1-3
- AUTOSAR C++14 Rule M3-2-1
- AUTOSAR C++14 Rule M3-2-2
- AUTOSAR C++14 Rule M3-2-4

Deactivated JSF C++ Coding Rules

The JSF C++ coding rules that are deactivated in the default Polyspace as You Code analysis include:

- JSF C++ Rule 46,137. See “Supported JSF C++ Coding Rules” on page 17-100.

Checkers with Reduced Scope in Polyspace as You Code

The checkers that finds fewer issues in the default Polyspace as You Code analysis are listed in the table. The issues that are not found are related to multiple-file analysis.

Checker	Behavior in the default Polyspace as You Code Analysis
CERT C: Rule EXP37-C	Does not check for “Function declaration mismatch”.
CERT C++: EXP37-C	Does not check for “Function declaration mismatch”.
CERT C++: DCL60-CPP	Does not check for “Nonidentical Definitions of Function or Object Across Modules”.
CERT C: Rec. DCL19-C	Does not check for “Function or object declared without static specifier and referenced in only one file”.
CERT C: Rec. DCL23-C	Does not check for “External identifiers not distinct”.
AUTOSAR C++14 Rule A0-1-3	Does not check for “Private Member Function Not Used”.
ISO/IEC TS 17961 [argcomp]	Does not check for “Conflicting declarations or conflicting declaration and definition”.

See Also

More About

- “Analysis Scope of Polyspace as You Code” on page 11-75
- “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 18-65

Troubleshoot Failed Analysis or Unexpected Results in Polyspace as You Code

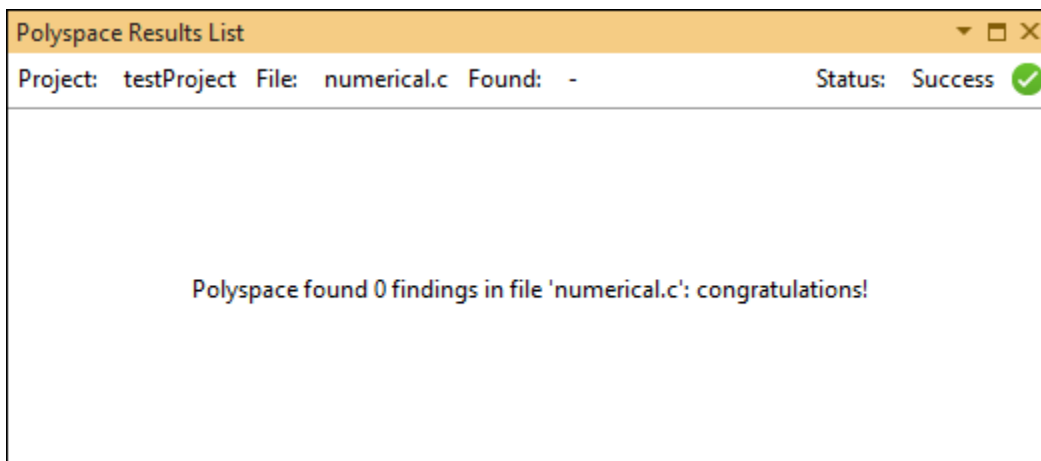
Issue

After installing and configuring Polyspace as You Code in your IDE, you should see analysis results as source code markers within a few seconds of starting the analysis (slightly longer for C++ files). If you do not see results, it could mean that the analysis did not find any issue or the analysis failed to complete or even failed to start.

If you run Polyspace as You Code on each save, some of the runs might fail because a file does not compile yet. If you do not see results despite successful compilation, you might have to investigate further and change the analysis options or extension settings. (Note that you can enable the checker `File does not compile` so that you always see at least one result, even when the file does not compile.)

Possible Solutions

If you do not see results, first confirm that the analysis reached completion. If the analysis completed but did not find any issue, on the IDE pane that shows the full list of results, you see a status message indicating completion. For instance, in Visual Studio, on the **Polyspace Results List** pane, you see this message:



If the analysis failed to complete, you also see a status message indicating failure. For further diagnosis, check the analysis log within the IDE. For instance, in Visual Studio, open the **Output** pane, select **Polyspace** from the **Show output from** drop-down list, and check the messages. You might have to scroll up a bit to see the root cause of the failure.

For more information on how to follow analysis progress in specific IDEs, see:

- “Run Polyspace as You Code in Visual Studio and Review Results” on page 29-2
- “Run Polyspace as You Code in Visual Studio Code and Review Results” on page 29-6
- “Run Polyspace as You Code in Eclipse and Review Results” on page 29-15

Check if Build Analysis is Outdated

The most common cause of an analysis failure is a compilation error. If a file compiles with your compiler but fails with Polyspace, it means that the analysis requires more information to emulate your compiler. In the most common scenario, the error indicates that you have to reanalyze your build.

If you specify in your extension settings that the analysis must use options extracted from a build command, a build task or a JSON compilation database, you must analyze your build command first and then run Polyspace as You Code. The build analysis must run on a command or task that performs a full build of your project or workspace and not an incremental build.

If you add a new file to your project or workspace but forget to rerun build analysis, you might see compilation errors when trying to analyze the new file. The most common error is that include files cannot be found. To fix the issue, simply rerun your build analysis and then run Polyspace as You Code. For details, see “Analyzing Build in Polyspace as You Code”.

Check for Mistakes in Options File

If you specify an options file in your extension settings, the analysis appends options from this file to the underlying run command. If an option is incorrectly written, for instance, refers to a nonexistent file or uses an incorrect argument, the analysis can fail.

You can see all errors and warnings related to options in the analysis log. To see a more detailed log, use the analysis option `-no-quiet`. You can enter this option in the same options file that contains your other options. See “Options Files for Polyspace Analysis” on page 12-5.

Check for Incorrect Path to Analysis Engine

In Visual Studio Code, you can change the extension setting that points to the Polyspace installation folder. If you enter an incorrect path in this setting, the Polyspace as You Code extension fails to start. You see a message indicating that the internal server, Polyspace Connector, attempted to start and then failed.

Check that the installation folder that you provided in your extension settings indeed contains a Polyspace installation. The path must contain a subfolder `polyspace\bin`, that contains the `polyspace-bug-finder-access` executable.

See Also

Related Examples

- “Run Polyspace as You Code in Visual Studio and Review Results” on page 29-2
- “Run Polyspace as You Code in Visual Studio Code and Review Results” on page 29-6
- “Run Polyspace as You Code in Eclipse and Review Results” on page 29-15
- “Run Polyspace as You Code from Command Line and Export Results” on page 29-21

Configuration Workflows Common to All Platforms

Configure Polyspace Analysis

- “Specify Polyspace Analysis Options” on page 12-2
- “Options Files for Polyspace Analysis” on page 12-5

Specify Polyspace Analysis Options

You can change the default options associated with a Polyspace analysis. For instance, you can:

- Change the set of defects that Bug Finder looks for.
See `Find defects (-checkers -disable-checkers)`.
- Change the default behavior of run-time checkers in Code Prover.

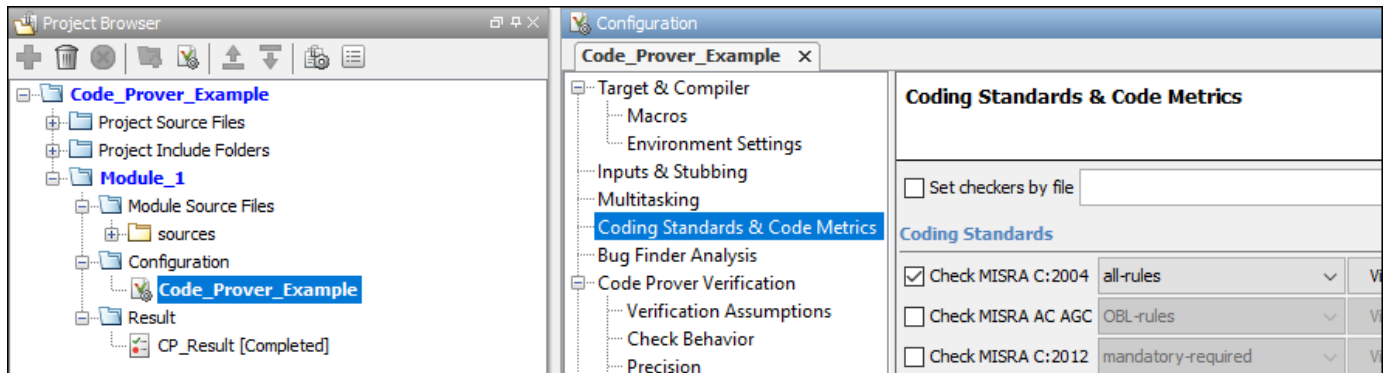
See, for instance, `Overflow mode for unsigned integer (-unsigned-integer-overflows)`.

For the full list of analysis options, see “Complete List of Polyspace Bug Finder Analysis Engine Options”.

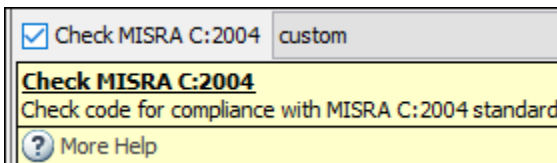
Depending on how you run Polyspace, you can configure the analysis options accordingly.

Polyspace User Interface

In the Polyspace user interface, you create a project for the analysis. The project can have one or more modules. Click the **Configuration** node in a module. On the **Configuration** pane, change options as needed.



For more information, see the tooltip on each option. Click the **More help** link for context-sensitive help on the options.



For more information, see “Run Analysis in Polyspace Desktop User Interface” on page 3-2.

Windows or Linux Scripts

Provide the options to the `polyspace-bug-finder` or `polyspace-code-prover` command. See also:

- `polyspace-bug-finder`
- `polyspace-code-prover`

For instance:

```
polyspace-code-prover -sources file_name \
    -main-generator main-generator-writes-variables all
```

You can also provide the options in a text file. See “Run Polyspace Analysis from Command Line” on page 4-2.

MATLAB Scripts

Create a `polyspace.Project` object and set the options through the `Configuration` property of the object. See also:

- `polyspace.Project`
- `polyspace.Project.Configuration` Properties

For instance:

```
proj = polyspace.Project;
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.BugFinderAnalysis.EnableCheckers = false;
```

See also “Run Polyspace Analysis by Using MATLAB Scripts” on page 5-9.

Eclipse and Eclipse-based IDEs

Select **Polyspace > Configure Project**. Set the options in the Configuration window.

Some Target & Compiler options are automatically extracted from your Eclipse project. See “Run Polyspace Analysis on Eclipse Projects” on page 10-2.

Simulink

In your Simulink model, specify the basic options through Simulink Configuration Parameters. On the **Apps** tab, select **Polyspace** and then on the **Polyspace** tab, select **Settings**.

To navigate to Polyspace analysis options related to the generated code, on the **Polyspace** tab, see **Settings > Project Settings**.

See:

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 6-2
- “Configure Polyspace Options in Simulink” on page 6-53

MATLAB Coder App

In the MATLAB Coder app, after code generation, specify the basic options through the **Polyspace** pane. From this window, you can navigate to the full set of Polyspace analysis options.

See:

- “Run Polyspace on C/C++ Code Generated from MATLAB Code” on page 7-2
- “Configure Advanced Polyspace Options in MATLAB Coder App” on page 7-7

Options Files for Polyspace Analysis

To adapt the Polyspace analysis configuration to your development environment and requirements, you have to modify the default configuration through command-line options such as `-compiler`. Options files are a convenient way to collect multiple options together and reuse them across projects.

What are Options Files

Options files are text files with one option per line. For instance, the content of an options file can look like this:

```
# Options for Polyspace analysis
# Options apply to all projects in Controller module
-compiler visual16.x
-D _WIN32
-code-behavior-specifications "Z:\utils\polyspace\forbiddenfunctions.xml"
```

The lines starting with `#` represent comments for better readability. These lines are ignored during analysis.

Specifying Options Files

Depending on the platform where you run analysis, you can specify an options file in one of the following ways.

Command Line

At the command line (and in scripts), specify an options file as argument to the option `-options-file`.

For instance, instead of the command:

```
polyspace-bug-finder -sources file.c -compiler visual16.x -D _WIN32
-code-behavior-specifications "Z:\utils\polyspace\forbiddenfunctions.xml"
```

Save this content:

```
-compiler visual16.x
-D _WIN32
-code-behavior-specifications "Z:\utils\polyspace\forbiddenfunctions.xml"
```

In a file `options.txt` in the path `Z:\utils\polyspace\` and shorten the command to:

```
polyspace-bug-finder -sources file.c -options-file "Z:\utils\polyspace\options.txt"
```

You can use options files with these Polyspace commands:

- `polyspace-bug-finder`
- `polyspace-bug-finder-server`
- `polyspace-bug-finder-access`
- `polyspace-code-prover`
- `polyspace-code-prover-server`

IDEs

If you run Polyspace as You Code using IDE extensions, you typically specify three groups of options differently:

Options Group	Extension Settings
<p><i>Build options:</i> You can extract build options from existing artifacts such as build commands and JSON compilation database.</p> <p>You can also collect all build options in an options file and specify the option file path in the appropriate extension setting.</p>	<ul style="list-style-type: none"> • Visual Studio Code — Analysis Options > Manual Setup > Build Setting : Polyspace Build Options File • Visual Studio — Get from Polyspace build options file (in section Build Configuration) • Eclipse — Get from Polyspace build options file (in section Build Configuration)
<p><i>Checkers:</i> You can select checkers using a checkers selection wizard.</p>	See “Setting Checkers in Polyspace as You Code”.
<p><i>Other remaining options:</i> All remaining options can be collected in a second options file that goes into the appropriate extension setting.</p>	<ul style="list-style-type: none"> • Visual Studio Code — Analysis Options > Manual Setup: Other Analysis Options • Visual Studio — Analysis configuration > Analysis options file • Eclipse — Analysis options file

If you use options files both for build options and other options, the result is the same as specifying a single options file with the other options appended to the build options. See also “Specifying Multiple Options Files” on page 12-6.

For more information on IDE extensions, see:

- “Configure Polyspace as You Code Extension in Visual Studio” on page 11-2
- “Configure Polyspace as You Code Extension in Visual Studio Code” on page 11-8
- “Configure Polyspace as You Code Plugin in Eclipse” on page 11-19

Polyspace User Interface

In the user interface of the Polyspace desktop products, you typically do not require an options file. Most options can be specified on the **Configuration** pane in the Polyspace user interface.

However, some options are available only at the command line and do not have a counterpart in the user interface. If you have to specify multiple command-line-only options, you can collect them in an options file, for instance `commandLineStyleOptions.txt`. On the **Configuration** pane, under the **Advanced Settings** node, specify the absolute path of the options file in the **Other** field:

```
-options-file C:\psconfig\commandLineStyleOptions.txt
```

Specifying Multiple Options Files

You can specify multiple options files in an analysis. For instance, at the command line, you can enter:

```
polyspace-bug-finder -sources file.c -options-file opts1.txt -options-file opts2.txt
```

When you specify multiple options files in an analysis, all options from the options files are appended to the analysis command. For instance, the preceding command has the same effect as using a single options file that places the content of `opts1.txt` above `opts2.txt`.

If an option appears in multiple files with conflicting arguments, the argument in the last options file prevails. For instance, in the preceding command, if `opts1.txt` contains:

```
-checkers all  
-misra3 all
```

And `opts2.txt` contains:

```
-misra3 single-unit-rules
```

The analysis uses only the argument `single-unit-rules` for the option `-misra3`.

You can use this stacking of options files to override options. For instance, suppose you use a read-only options file that applies to your entire team but want to override some of the options in the file. You can override the options by using a second options file that you create and specifying your options file *after* the team-wide options file.

You can also specify the option `-options-file` within an options file and aggregate several options files in this way.

See Also

`-options-file`

Related Examples

- “Run Polyspace Analysis from Command Line” on page 4-2
- “Run Polyspace Bug Finder on Server and Upload Results to Web Interface”
- “Run Polyspace as You Code from Command Line and Export Results” on page 29-21
- “Complete List of Polyspace Bug Finder Analysis Engine Options”

Configure Target and Compiler Options

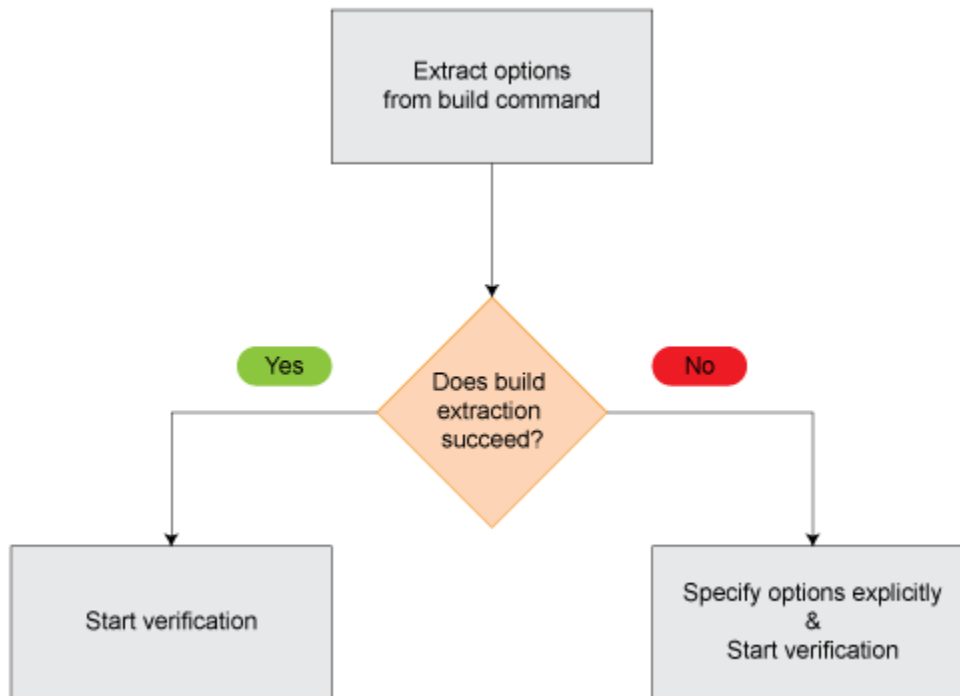
Specify Target Environment and Compiler Behavior

Before verification, specify your source code language (C or C++), target processor, and the compiler that you use for building your code. In certain cases, to emulate your compiler behavior, you might have to specify additional options.

Using your specification, the verification determines the sizes of fundamental types, considers certain macros as defined, and interprets compiler-specific extensions of the Standard. If the options do not correspond to your run-time environment, you can encounter:

- Compilation errors
- Verification results that might not apply to your target

If you use a build command such as `gmake` to build your code and the build command meets certain restrictions, you can extract the options from the build command after executing the command. Otherwise, specify the options explicitly.



Extract Options from Build Command

If you use build automation scripts to build your source code, you can set up a Polyspace project from your scripts. The options associated with your compiler are specified in that project.

In the Polyspace desktop products, for information on how to trace your build command from the:

- Polyspace user interface, see "Add Source Files for Analysis in Polyspace Desktop User Interface" on page 2-2.
- DOS or UNIX command line, see `polyspace-configure`.

- MATLAB command line, see `polyspaceConfigure`.

In the Polyspace server products, for information on how to trace your build command, see “Create Polyspace Analysis Configuration from Build Command (Makefile)” on page 13-22.

For Polyspace project creation, your build automation script (makefile) must meet certain requirements. See “Requirements for Project Creation from Build Systems” on page 13-24.

Specify Options Explicitly

If you cannot trace your build command and therefore manually create a project, you have to specify the options explicitly.

- In the user interface of the Polyspace desktop products, select a project configuration. On the **Configuration** pane, select **Target & Compiler**. Specify the options.
- At the DOS or UNIX command line, specify flags with the `polyspace-bug-finder`, `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server` command.
- At the MATLAB command line, specify arguments with the `polyspaceBugFinder`, `polyspaceCodeProver`, `polyspaceBugFinderServer` or `polyspaceCodeProverServer` function.

Specify the options in this order.

- Required options:
 - **Source code language (-lang)**: If all files have the same extension `.c` or `.cpp`, the verification uses the extension to determine the source code language. Otherwise, explicitly specify the option.
 - **Compiler (-compiler)**: Select the compiler that you use for building your source code. If you cannot find your compiler, use an option that closely matches your compiler.
 - **Target processor type (-target)**: Specify the target processor on which you intend to execute your code. For some processors, you can change the default specifications. For instance, for the processor `hc08`, you can change the size of types `double` and `long double` from 32 to 64 bits.

If you cannot find your target processor, you can create your own target and specify the sizes of fundamental types, default signedness of `char`, and endianness of the target machine. See **Generic target options**.

- Language-specific options:
 - **C standard version (-c-version)**: The default C language standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C99 standard. Specify an earlier standard such as C90 or a later standard such as C11.
 - **C++ standard version (-cpp-version)**: The default C++ language standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C++03 standard. Specify later standards such as C++11 or C++14.
- Compiler-specific options:

Whether these options are available or not depends on your specification for **Compiler (-compiler)**. For instance, if you select a `visual` compiler, the option `Pack alignment value`

(`-pack-alignment-value`) is available. Using the option, you emulate the compiler option `/Zp` that you use in Visual Studio.

For all compiler-specific options, see “Target and Compiler”.

- Advanced options:

Using these options, you can modify the verification results. For instance, if you use the option `Division round down (-div-round-down)`, the verification considers that quotients from division or modulus of negative numbers are rounded down. Use these options only if you use similar options when compiling your code.

For all advanced options, see “Target and Compiler”.

- Compiler header files:

If you specify the `diab`, `tasking` or `greenhills` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis” on page 13-20.

If you still see compilation errors after running analysis, you might have to specify other options:

- *Define macros*: Sometimes, a compilation error occurs because the analysis considers a macro as undefined. Explicitly define these macros. See `Preprocessor definitions (-D)`.
- *Specify include files*: Sometimes, a compilation error occurs because your compiler defines standard library functions differently from Polyspace and you do not provide your compiler include files. Explicitly specify the path to your compiler include files. See “Provide Standard Library Headers for Polyspace Analysis” on page 13-20.

See Also

`Source code language (-lang)` | `Compiler (-compiler)` | `Target processor type (-target)` | `C standard version (-c-version)` | `C++ standard version (-cpp-version)` | `Preprocessor definitions (-D)`

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 13-5
- “Provide Standard Library Headers for Polyspace Analysis” on page 13-20

C/C++ Language Standard Used in Polyspace Analysis

The Polyspace analysis adheres to a specific language standard for code compilation. The language standard, along with your compiler specification, defines the language elements that you can use in your code. For instance, if the Polyspace analysis uses the C99 standard, C11 features such as use of the thread support library from `threads.h` causes compilation errors.

Supported Language Standards

The Polyspace analysis supports these standards:

- **C:** C90, C99, C11, C17

The default standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C99 standard. To change the language standard, use the option `C standard version (-c-version)`.

- **C++:** C++03, C++11, C++14

The default standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C++03 standard. To change the language standard, use the option `C++ standard version (-cpp-version)`.

Default Language Standard

The default language standard depends on your specification for the option `Compiler (-compiler)`.

GCC compilers

Compiler	C Standard	C++ Standard
gnu3.4, gnu4.6, gnu4.7, gnu4.8, gnu4.9	C99	C++03
gnu5.x	C11	C++03
gnu6.x	C11	C++14
gnu7.x	C11	C++14
gnu8.x	C17	C++14
gnu9.x	C17	C++14
gnu10.x	C17	C++14
gnu11.x	C17	C++17
gnu12.x	C17	C++17

Clang compilers

Compiler	C Standard	C++ Standard
clang3.x	C99	C++03 The analysis accepts some C++11 extensions.
clang4.x	C99	C++03 The analysis accepts C++14 extensions.
clang5.x	C99	C++03 The analysis accepts C++14 extensions.
clang6.x	C99	C++14
clang7.x	C99	C++14
clang8.x	C99	C++14
clang9.x	C99	C++14
clang10.x	C99	C++14
clang11.x	C17	C++14
clang12.x	C17	C++14
clang13.x	C17	C++14

Visual Studio compilers

Compiler	C Standard	C++ Standard
visual9.0	C99	C++03
visual10.0		
visual11.0		
visual12.0		
visual14.0	C99	C++14
visual15.x	C99	C++14
visual16.x	C99	C++14

Other Compilers

Compiler	C Standard	C++ Standard
armcc	C99	C++03
armclang	C11	C++03
codewarrior	C99	C++03
cosmic	C99	Not supported

Compiler	C Standard	C++ Standard
diab	C99	C++03
generic	C99	C++03
greenhills	C99	C++03
iar	C99	C++03
iar-ew	C99	C++03
intel	C11	C++14
keil	C99	C++03
microchip	C99	Not supported
renesas	C99	C++03
tasking	C99	C++03
ti	C99	C++03

See Also

Compiler (-compiler) | C standard version (-c-version) | C++ standard version (-cpp-version)

More About

- “C11 Language Elements Supported in Polyspace” on page 13-8
- “C++11 Language Elements Supported in Polyspace” on page 13-10
- “C++14 Language Elements Supported in Polyspace” on page 13-13
- “C++17 Language Elements Supported in Polyspace” on page 13-16

C11 Language Elements Supported in Polyspace

This table provides a partial list of C language elements that have been introduced since C11 and the corresponding Polyspace support. If your code contains non-supported constructions, Polyspace reports a compilation error.

C11 Language Element	Supported
<code>alignas</code> and <code>alignof</code> convenience macros	Yes
<code>aligned_alloc</code> function	Yes
<code>noreturn</code> convenience macros	Yes
Generic selection	Yes
Thread support library (<code>threads.h</code>)	Yes
Atomic operations library (<code>stdatomic.h</code>)	Yes
Atomic types with <code>_Atomic</code>	Yes. If you use the Clang compiler, see limitations book for limitations on atomic data types. See “Limitations of Polyspace Verification” (Polyspace Code Prover).
UTF-16 and UTF-32 character utilities	Yes
Bound-checking interfaces or alternative versions of standard library functions that check for buffer overflows (Annex K of C11) For instance, <code>strcpy_s</code> is an alternative to <code>strcpy</code> that checks for certain errors in the string copy.	No. Polyspace checks for certain run-time errors in use of standard library functions. The checking does not extend to these alternatives.
Anonymous structures and unions	Yes
Static assert declaration	Yes
Features related to error handling such as <code>errno_t</code> and <code>rsize_t</code> typedef-s	No. If you see compilation errors from use of these typedef-s, explicitly specify the path to your compiler headers. See “Provide Standard Library Headers for Polyspace Analysis” on page 13-20.
<code>quick_exit</code> and <code>at_quick_exit</code>	Yes. In Bug Finder, functions registered with <code>at_quick_exit</code> appear as uncalled.
<code>CMPLX</code> , <code>CMPLXF</code> and <code>CMPLXL</code> macros	Yes

See Also

C standard version (`-c-version`)

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 13-5

C++11 Language Elements Supported in Polyspace

This table provides a partial list of C++ language elements that have been introduced since C++11 and its corresponding Polyspace support. If your code contains nonsupported constructions, Polyspace reports a compilation error.

C++11 Std Ref	Description	Supported
C++2011-DR226	Default template arguments for function templates	Yes
C++2011-DR339	Solving the SFINAE problem for expressions	Yes
C++2011-N1610	Initialization of class objects by rvalues	Yes
C++2011-N1653	C99 preprocessor	Yes
C++2011-N1720	Static assertions	Yes
C++2011-N1737	Multi-declarator auto	Yes
C++2011-N1757	Right angle brackets	Yes
C++2011-N1791	Extended friend declarations	No
C++2011-N1811	long long	Yes
C++2011-N1984	auto-typed variables	Yes
C++2011-N1986	Delegating constructors	Yes
C++2011-N1987	Extern templates	Yes
C++2011-N1988	Extended integral types	Yes
C++2011-N2118	Rvalue references	Yes
C++2011-N2170	Universal character name literals	Yes
C++2011-N2179	Concurrency: Propagating exceptions	No
C++2011-N2235	Generalized constant expressions	Yes
C++2011-N2239	Concurrency: Sequence points	No new syntax/ keyword is introduced and therefore does not affect Polyspace support for C++11.
C++2011-N2242	Variadic templates	Yes
C++2011-N2249	New character types	Yes
C++2011-N2253	Extending sizeof	Yes
C++2011-N2258	Template aliases	Yes
C++2011-N2340	<code>__func__</code> predefined identifier	Yes
C++2011-N2341	Alignment support	Yes
C++2011-N2342	Standard Layout Types	Yes
C++2011-N2343	Declared type of an expression	Yes
C++2011-N2346	Defaulted and deleted functions	Yes
C++2011-N2347	Strongly typed enums	Yes

C++11 Std Ref	Description	Supported
C++2011-N2427	Concurrency: Atomic operations	No
C++2011-N2429	Concurrency: Memory model	No new syntax/ keyword is introduced and therefore does not affect Polyspace support for C++11.
C++2011-N2431	Null pointer constant	Yes
C++2011-N2437	Explicit conversion operators	Yes
C++2011-N2439	Rvalue references for *this	Yes
C++2011-N2440	Concurrency: Abandoning a process and at_quick_exit	Yes
C++2011-N2442	Unicode string literals	Yes
C++2011-N2442	Raw string literals	Yes
C++2011-N2535	Inline namespaces	Yes
C++2011-N2540	Inheriting constructors	Yes
C++2011-N2541	New function declarator syntax	Yes
C++2011-N2544	Unrestricted unions	Yes
C++2011-N2546	Removal of auto as a storage-class specifier	Yes
C++2011-N2547	Concurrency: Allow atomics use in signal handlers	No
C++2011-N2555	Extending variadic template template parameters	Yes
C++2011-N2657	Local and unnamed types as template arguments	Yes
C++2011-N2659	Concurrency: Thread-local storage	No
C++2011-N2660	Concurrency: Dynamic initialization and destruction with concurrency	Yes
C++2011-N2664	Concurrency: Data-dependency ordering: atomics and memory model	No
C++2011-N2672	Initializer lists	Yes
C++2011-N2748	Concurrency: Strong Compare and Exchange	No
C++2011-N2752	Concurrency: Bidirectional Fences	No
C++2011-N2756	Nonstatic data member initializers	Yes
C++2011-N2761	Generalized attributes	Yes
C++2011-N2764	Forward declarations for enums	Yes
C++2011-N2765	User-defined literals	Yes
C++2011-N2927	New wording for C++0x lambdas	Yes
C++2011-N2928	Explicit virtual overrides	Yes
C++2011-N2930	Range-based for	Yes
C++2011-N3050	Allowing move constructors to throw [noexcept]	Yes
C++2011-N3053	Defining move special member functions	Yes

C++11 Std Ref	Description	Supported
C++2011-N3276	decltype and call expressions	Yes

See Also

C++ standard version (-cpp-version)

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 13-5
- “C++14 Language Elements Supported in Polyspace” on page 13-13
- “C++17 Language Elements Supported in Polyspace” on page 13-16

C++14 Language Elements Supported in Polyspace

This table provides a partial list of C++ language elements that have been introduced since C++14 and its corresponding Polyspace support. If your code contains nonsupported constructions, Polyspace reports a compilation error.

C++14 Std Ref	Description	Supported
C++2014-N3323	Implicit conversion from class type in certain contexts such as <code>delete</code> or <code>switch</code> statement.	This C++14 feature allows implicit conversion from class type in certain contexts. No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3462	More SFINAE-friendly <code>std::result_of</code>	Yes
C++2014-N3472	Binary literals, for instance, <code>0b100</code> .	Yes
C++2014-N3545	<code>operator()</code> in <code>integral_constant</code> template of <code>constexpr</code> type	Yes
C++2014-N3637	Relation between <code>std::async</code> and destructor of <code>std::future</code>	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3638	Automatic deduction of return type for functions where an explicit return type is not specified	Yes. In some cases, Code Prover can show compilation errors.
C++2014-N3642	Suffixes for user-defined literals indicating time (<code>h</code> , <code>min</code> , <code>s</code> , <code>ms</code> , <code>us</code> , <code>ns</code>) and strings (<code>s</code>)	Yes
C++2014-N3648	Initialization of captured members in lambda functions	Yes. In some cases, during initialization, Code Prover can call the corresponding constructors more number of times than necessary.
C++2014-N3649	Generic (polymorphic) lambda expressions: <ul style="list-style-type: none"> Using <code>auto</code> type-specifier for parameter and return type Conversion of generic capture-less lambda expressions to pointer-to-function. 	Yes

C++14 Std Ref	Description	Supported
C++2014-N3651	Variable templates	Yes
C++2014-N3652	Declarations, conditions and loops in <code>constexpr</code> functions.	Yes
C++2014-N3653	<p>Initialization of aggregate classes with fewer initializers than members</p> <p>For instance, this initialization has fewer initializers than members. The member <code>c</code> is initialized with the value 0 and <code>d</code> is initialized with the value <code>s</code>.</p> <pre>struct S { int a; const char* b; int c; int d = b[a];}; S ss = { 1, "asdf" };</pre>	Yes
C++2014-N3654	<code>std::quoted</code>	Yes
C++2014-N3656	<code>std::make_unique</code>	Yes
C++2014-N3658	<code>std::integer_sequence</code>	Yes
C++2014-N3658	<code>std::shared_lock</code>	No. The use of <code>std::shared_lock</code> does not cause compilation errors but the construct is not semantically supported.
C++2014-N3664	Calling <code>new</code> and <code>delete</code> operators in batches.	This C++14 feature clarifies how successive calls to the <code>new</code> operator are implemented. No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3668	<code>std::exchange</code>	Partially supported.
C++2014-N3670	Using <code>std::get</code> with a data type to get one element in an <code>std::tuple</code> (provided there is only one element of the type in the tuple)	Yes
C++2014-N3671	Overloads for <code>std::equal</code> , <code>std::mismatch</code> and <code>std::is_permutation</code> function templates that accept two separate ranges	Yes
C++2014-N3733	Removal of <code>std::gets</code> from <code><cstdio></code>	Yes

C++14 Std Ref	Description	Supported
C++2014-N3776	Wording change for destructor of <code>std::future</code>	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3779	<code>std::complex</code> literals representing pure imaginary numbers with suffix <code>i</code> , <code>if</code> or <code>il</code>	Yes
C++2014-N3781	Use of single quotation mark as digit separator, for instance, <code>1'000</code> .	Yes
C++2014-N3786	Prohibiting "out of thin air" results in C++14	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3910	Synchronizing behavior of signal handlers	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3924	Discouraging use of <code>rand()</code>	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3927	Lock-free executions	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.

See Also

C++ standard version (`-cpp-version`)

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 13-5
- “C++11 Language Elements Supported in Polyspace” on page 13-10
- “C++17 Language Elements Supported in Polyspace” on page 13-16

C++17 Language Elements Supported in Polyspace

This table provides a partial list of C++ language elements that have been introduced since C++17 and its corresponding Polyspace support. If your code contains nonsupported constructions, Polyspace reports a compilation error.

C++17 Std Ref	Description	Supported
C++2017-N3921	<code>std::string-view</code> : Observe the content of an <code>std::string</code> object without owning the resource	Yes
C++2017-N3922	<ul style="list-style-type: none"> When used in copy-list-initialization, <code>auto</code> deduces the type to be an <code>std::initializer_list</code> if the elements of the initializer list have an identical type. Otherwise, the <code>auto</code> deduction is ill-formed. When using direct list-initialization with a braced initializer list containing a single element, <code>auto</code> deduces the type from that element. When using direct list-initialization with a braced initializer list containing more than a single element, <code>auto</code> deduction of type is ill-formed. 	Yes
C++2017-N3928	The <code>static_assert</code> declaration no longer requires a second argument. Invoking <code>static_assert</code> with no message is now allowed: <code>static_assert(N > 0);</code>	Yes
C++2017-N4051	C++ has templates that are not class templates, such as a template that takes templates as an argument. Previously, declaring such template-template parameters required the use of the <code>class</code> keyword. In C++17, you can use <code>typename</code> when declaring template-template parameters, such as: <code>template <template <typename> typename Tmpl> struct X;</code>	Yes
C++2017-N4086	Starting in C++17, trigraphs are no longer supported.	No
C++2017-N4230	Starting in C++17, use a qualified name in a namespace definition to define several nested namespaces at once. For instance, these code snippets are equivalent: <ul style="list-style-type: none"> <code>namespace base::derived{ //.. }</code> <code>namespace { namespace derived{ //... } }</code> 	Yes

C++17 Std Ref	Description	Supported
C++2017-N4259	The function <code>std::uncaught_exceptions</code> is introduced in C++17, which returns the number of exceptions in your code that are not handled. The function <code>std::uncaught_exception</code> , which returns a Boolean value, is deprecated.	Yes
C++2017-N4266	Starting in C++17, namespaces and enumerators can be annotated with attributes to allow clearer communication of developer intention.	Yes
C++2017-N4267	Starting in C++17, the prefix <code>u8</code> is supported. This prefix creates a UTF-8 character literal. The value of the UTF-8 character literal is equal to its ISO 10646 code point value if the code point value is in the C0 Controls and Basic Latin Unicode block.	Yes
C++2017-N4268	Allow constant evaluation of nontype template arguments.	Yes
C++2017-N4295	Allow fold expressions	Yes
C++2017-N4508	Allow untyped <code>std::shared_mutex</code>	The use of <code>std::shared_mutex</code> does not cause a compilation error. Polyspace does not support sharing mutex objects by using <code>std::shared_mutex</code> .
C++2017-P0001R1	Remove the use of the <code>register</code> keyword	Yes
C++2017-P0002R1	Remove <code>operator++(bool)</code>	Yes
C++2017-P0003R5	Remove deprecated exception specifications by using <code>throw(<>)</code>	Bug Finder removes the exception specification specified by using <code>throw()</code> statements. Code Prover raises a compilation error when <code>throw()</code> statements are present in C++17 code.
C++2017-P0012R1	Make exception specifications part of the type system	Yes
C++2017-P0017R1	Aggregate initialization of classes with base classes	Yes
C++2017-P0018R3	Allow capturing the pointer <code>*this</code> in Lambda expressions	Yes
C++2017-P0024R2	Standardization of the C++ technical specification for Extension for Parallelism	Polyspace supports this feature when you use the Visual 15.x and Intel C++ 18.0 compilers.

C++17 Std Ref	Description	Supported
C++2017-P002842	Using attribute namespaces without repetition	Yes
C++2017-P0035R4	Dynamic memory allocation for over-aligned data	Yes
C++2017-P0036R0	Unary fold expressions and empty parameter packs	Yes
C++2017-P0061R1	Use of <code>__has_include</code> in preprocessor conditionals	Yes
C++2017-P0067R5	Elementary string conversions	No
C++2017-P0083R3	Splicing maps and sets	Polyspace supports this feature when the compiler you use also supports this feature. For instance, Polyspace supports this feature when you use g++ as compiler.
C++2017-P0088R3	<code>std::variant</code>	Partially supported.
C++2017-P0091R3	Template argument deduction for class templates	Partially supported.
C++2017-P0127R2	Non-type template parameters that have auto type	Yes
C++2017-P0135R1	Guaranteed copy elision	Partially supported.
C++2017-P0136R1	New specification for inheriting constructors	No
C++2017-P0137R1	Replacement of class objects containing reference members	Yes
C++2017-P0138R2	Direct-list-initialization of enumerations	Yes
C++2017-P0145R3	Stricter expression evaluation order	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++17.
C++2017-P0154R1	Hardware interference size	Supported with Visual Studio Compiler
C++2017-P0170R1	<code>constexpr</code> Lambda expressions	Partially supported
C++2017-P018R0	Differing begin and end types in range-based for loops	Yes
C++2017-P0188R1	<code>[[fallthrough]]</code> attribute	Yes

C++17 Std Ref	Description	Supported
C++2017-P0189R1	[[nodiscard]] attribute	Yes
C++2017-P0195R2	Pack expansions in using-declarations	Yes
C++2017-P0212R1	[[maybe_unused]] attribute	Yes
C++2017-P0217R3	Structured Bindings	Polyspace does not support binding by using an rvalue.
C++2017-P0218R1	std::filesystem	No
C++2017-P0220R1	std::any	Yes
C++2017-P0220R1	std::optional	Bug Finder supports the syntax. The semantics are partially supported. Code Prover does not support this feature.
C++2017-P0226R1	Mathematical special functions	No
C++2017-P0245R1	Hexadecimal floating-point literals	Yes
C++2017-P0283R2	Ignore unknown attributes	Yes
C++2017-P0292R2	constexpr if statements	Yes
C++2017-P0298R3	std::byte	Yes
C++2017-P0305R1	init-statements for if and switch	Yes
C++2017-P0386R2	Inline variables	No
C++2017-P0522R0	Invoke partial ordering to determine when a template <i>template-argument</i> is a valid match for a <i>template-parameter</i>	Partially supported

See Also

C++ standard version (-cpp-version)

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 13-5
- “C++11 Language Elements Supported in Polyspace” on page 13-10
- “C++14 Language Elements Supported in Polyspace” on page 13-13

Provide Standard Library Headers for Polyspace Analysis

Before Polyspace analyzes the code for bugs and run-time errors, it compiles your code. Even if the code compiles with your compiler, you can see compilation errors with Polyspace. If the error comes from a standard library function, it usually indicates that Polyspace is not using your compiler headers. To work around the errors, provide the path to your compiler headers.

If you create a Polyspace project or options file from your build command using `polyspace-configure`, the header paths are automatically added to this project or options file. Otherwise, you have to explicitly add these paths. This topic shows how to locate the standard library headers from your compiler. The code examples cause a compilation error that shows the location of the headers.

- To locate the folder containing your C compiler system headers, compile this C code by using your compilation toolchain:

```
float fopen(float f);
#include <stdio.h>
```

The code does not compile because the `fopen` declaration conflicts with the declaration inside `stdio.h`. The compilation error shows the location of your compiler implementation of `stdio.h`. Your C standard library headers are all likely to be in that folder.

- To locate the folder containing your C++ compiler system headers, compile this C++ code by using your compilation toolchain:

```
namespace std {
    float cin;
}
#include <iostream>
```

The code does not compile because the `cin` declaration conflicts with the declaration inside `iostream.h`. The compilation error shows the location of your compiler implementation of `iostream.h`. Your C++ standard library headers are all likely to be in that folder.

After you locate the path to your compiler's header files, specify the path for the Polyspace analysis. For C++ code, specify the paths to both your C and C++ headers.

- In the user interface (Polyspace desktop products), add the folder to your project.

For more information, see “Add Source Files for Analysis in Polyspace Desktop User Interface” on page 2-2.

- At the command line, use the flag `-I` with one of these commands:

- `polyspace-bug-finder`
- `polyspace-bug-finder-server`
- `polyspace-code-prover`
- `polyspace-code-prover-server`

For more information, see `-I`.

See Also

More About

- “Fix Errors from Use of Polyspace Header Files” on page 32-42

Create Polyspace Analysis Configuration from Build Command (Makefile)

To run Polyspace with scripts at regular intervals, for instance, on a server during continuous integration, you must configure all analysis options beforehand so that the analysis completes without errors. These options must be updated as necessary to keep up with new code submissions. If you use existing artifacts such as a build command (makefile) to build new code submissions, you can reuse the build command to configure a Polyspace analysis and stay updated with new submissions. With the `polyspace-configure` command, you can monitor the execution of a build command and create an options file for analysis with Polyspace.

This topic shows a simple tutorial illustrating how to create an options file from a build command and use the file for the subsequent analysis. The topic uses a Linux makefile and the GCC compiler, but you can adapt the commands to other operating systems such as Windows and other compilers such as Microsoft Visual Studio.

- 1 Copy the demo source files from `polyspaceserverroot\polyspace\examples\cxx\Bug_Finder_Example\sources` to a folder with write permissions. Here, `polyspaceserverroot` is the root installation folder of the Polyspace server products, for instance, `C:\Program Files\Polyspace Server\R2019a`.
- 2 Create a simple makefile that compiles the demo source files. Save the makefile in the same folder as the source files.

For instance, create a file named `makefile` and add this content:

```
CC := gcc
SOURCES := $(wildcard *.c)

all: $(CC) -c $(SOURCES)
```

Check that the makefile builds the source files successfully. Open a command terminal, navigate to the folder (using `cd`) and enter:

```
make -B
```

The `make` command should complete execution without errors.

The `-B` option ensures that all targets in the makefile are built. Typically, build commands such as `make` are set up to only build sources that have changed since the previous build. However, `polyspace-configure` requires a full build to determine which sources to add to a Polyspace project or options file.

- 3 Trace the build command with `polyspace-configure` and create an options file `compile_opts`.

```
polyspace-configure -output-options-file compile_opts make -B
```

- 4 Create a second options file with additional options. For instance, create a file `run_opts` with this content:

```
-checkers numerical
-report-template BugFinder
-output-format pdf
```

The options run all numerical checkers in Bug Finder and creates a PDF report after analysis using the BugFinder template.

- 5 Run a Bug Finder analysis with the two options files: `compile_opts` created from your build command and `run_opts` created manually.

Polyspace Bug Finder:

```
polyspace-bug-finder -options-file compile_opts -options-file run_opts
```

Polyspace Bug Finder Server:

```
polyspace-bug-finder-server -options-file compile_opts -options-file run_opts
```

The analysis should complete without errors. You can open the results in the Polyspace user interface or upload the results to the Polyspace Access web interface (using the `polyspace-access` command).

To run Code Prover instead of Bug Finder, use the `polyspace-code-prover-server` command instead of the `polyspace-bug-finder-server` command.

You can run a similar analysis using MATLAB scripts. Replace `polyspace-bug-finder` with the `polyspaceBugFinder` function and `polyspace-configure` with the function `polyspaceConfigure`.

See Also

`polyspace-configure` | `polyspace-bug-finder-server`

See Also

More About

- “Specify Target Environment and Compiler Behavior” on page 13-2
- “Select Files for Polyspace Analysis Using Pattern Matching” on page 4-11
- “Modularize Polyspace Analysis by Using Build Command” on page 4-5

Requirements for Project Creation from Build Systems

For automatic project creation from build systems, your build commands or makefiles must meet certain requirements.

Compiler Requirements

- Your compiler must be called locally.

If you use a compiler cache such as `ccache` or a distributed build system such as `distmake`, the software cannot trace your build. You must deactivate them.

- Your compiler must perform a clean build.

If your compiler performs only an incremental build, use appropriate options to build all your source files. For example, if you use `gmake`, append the `-B` or `-W makefileName` option to force a clean build. For the list of options allowed with the GNU® `make`, see `make options`.

- Your compiler configuration must be available to Polyspace. The compilers currently supported include the following:

- Arm Keil.
- ARM® v5. See also `ARM v5 Compiler (-compiler armcc)`.
- ARM v6. See also `ARM v6 Compiler (-compiler armclang)`.
- Clang. For a list of supported versions, see “Clang Compilers”.
- Cosmic. See also `Cosmic Compiler (-compiler cosmic)`.
- Wind River® Diab. See also `Diab Compiler (-compiler diab)`.
- Green Hills®. See also `Green Hills Compiler (-compiler greenhills)`.
- GNU C/C++. For a list of supported versions, see “GCC Compilers”.
- IAR Embedded Workbench. See also `IAR Embedded Workbench Compiler (-compiler iar-ew)`.
- IAR systems.
- Intel® C++ Compiler Classic (`icc/icl`) compiler. See also `Intel C++ Compiler Classic (icc/icl) (-compiler intel)`.
- Microsoft Visual C++®. For a list of supported versions, see “Visual Studio Compilers”.
- MPLAB XC8 C. See also `MPLAB XC8 C Compiler (-compiler microchip)`.
- NXP CodeWarrior®. See also `NXP CodeWarrior Compiler (-compiler codewarrior)`.
- Renesas®. See also `Renesas Compiler (-compiler renesas)`.
- Altium® Tasking. See also `TASKING Compiler (-compiler tasking)`.
- Texas Instruments®. See also `Texas Instruments Compiler (-compiler ti)`.
- `tcc` - Tiny C Compiler

If your compiler configuration is not available to Polyspace:

- Write a compiler configuration file for your compiler in a specific format. For more information, see “Create Polyspace Projects from Build Systems That Use Unsupported Compilers” on page 32-11.

- Contact MathWorks Technical Support. For more information, see “Contact Technical Support About Issues with Running Polyspace” on page 32-6.
- If you build your code in Cygwin™, use versions 2.x or 3.x of Cygwin for Polyspace project creation from your build system (for instance, Cygwin version 2.10 or 3.0).
- With the TASKING compiler, if you use an alternative sfr file with extension `.asfr`, Polyspace might not be able to locate your file. If you encounter an error, explicitly `#include` your `.asfr` file in the preprocessed code using the option `Include (-include)`.

Typically, you use the statement `#include __SFRFILE__(__CPU__)` along with the compiler option `--alternative-sfr-file` to specify an alternative sfr file. The path to the file is typically `Tasking_C166_INSTALL_DIR\include\sfr\regCPUNAME.asfr`. For instance, if your TASKING compiler is installed in `C:\Program Files\Tasking\C166-VX_v4.0r1\` and you use the CPU-related flag `-Cxc2287m_104f` or `--cpu=xc2287m_104f`, the path is `C:\Program Files\Tasking\C166-VX_v4.0r1\include\sfr\regxc2287m.asfr`.

Build Command Requirements

- Your build command must run to completion without any user interaction.
- In Linux, only UNIX shell (sh) commands must be used. If your build uses advanced commands such as commands supported only by bash, tcsh or zsh, Polyspace cannot trace your build.

In Windows, only DOS commands must be used. If your build uses advanced commands such as commands supported only by PowerShell or Cygwin, Polyspace cannot trace your build. To see if Polyspace supports your build command, run the command from `cmd.exe` in Windows. For more information, see “Check if Polyspace Supports Build Scripts” on page 32-18.

- Your build command must not contain lines where several sources are compiled in a single line using wildcard characters, for instance:

```
cl.exe *.c
```

- If you use statically linked libraries, Polyspace cannot trace your build. In Linux, you can install the full Linux Standard Base (LSB) package to allow dynamic linking. For example, on Debian® systems, install LSB with the command `apt-get install lsb`.
- Your build command must not use aliases.

The `alias` command is used in Linux to create an alternate name for commands. If your build command uses those alternate names, Polyspace cannot recognize them.

- Your build process must not use the `LD_PRELOAD` mechanism.
- Your build command must be executable completely on the current machine and must not require privileges of another user.

If your build uses `sudo` to change user privileges or `ssh` to remotely log in to another machine, Polyspace cannot trace your build.

- If your build command uses redirection with the `>` or `|` character, the redirection occurs after Polyspace traces the command. Therefore, Polyspace does not handle the redirection.

For example, if your command occurs as

```
command1 | command2
```

And you enter

```
polyspace-configure command1 | command2
```

When tracing the build, Polyspace traces the first command only.

- If the System Integrity Protection (SIP) feature is active on the operating system macOS El Capitan (10.11) or a later macOS version, you can use `polyspace-configure` to generate a project or build options file only if your build system supports the generation of a compilation database file that you pass to `polyspace-configure`. See “Create Polyspace Options File from JSON Compilation Database”. If your build system does not support the generation of compilation database file and the SIP feature is enabled, Polyspace cannot trace your build command. Alternatively, before tracing your build command, disable the SIP feature. You can reenable this feature after tracing the build command.

Similar considerations apply to other security applications such as security-related products from CylanceProtect, Avecto and Tanium.

- If your computer hibernates during the build process, Polyspace might not be able to trace your build.
- When creating projects from build commands in the Polyspace User Interface, you might encounter errors such as `libcurl.so.4: version 'CURL_OPENSSL_3' not found`. In such cases, create the Polyspace project by using the command `polyspace-configure` in the system command line interface, using the build command as the argument. See `polyspace-configure`.

Note Your environment variables are preserved when Polyspace traces your build command.

See Also

`polyspace-configure`

Related Examples

- “Add Source Files for Analysis in Polyspace Desktop User Interface” on page 2-2
- “Create Polyspace Analysis Configuration from Build Command (Makefile)” on page 13-22

Supported Keil or IAR Language Extensions

Polyspace analysis can interpret a subset of common C/C++ language constructs and extended keywords by default. For compiler-specific keywords, you must specify your choice of compiler. If you specify `keil` or `iar` for `Compiler` (`-compiler`), the Polyspace verification allows language extensions specific to the Keil or IAR compilers.

Special Function Register Data Type

Embedded control applications frequently read and write port data, set timer registers, and read input captures. To deal with these requirements without using assembly language, some microprocessor compilers define special data types such as `sfr` and `sbit`. Typical declarations are:

```
sfr A0 = 0x80;
sfr A1 = 0x81;
sfr ADCUP = 0xDE;
sbit EI = 0x80;
```

The declarations reside in header files such as `regxx.h` for the basic 80Cxxx micro processor. The declarations customize the compiler to the target processor.

You access a register or a port by using the `sfr` and `sbit` data as follows. However, these data types are not part of the C99 Standard.

```
int status,P0;

void main (void) {
    ADCUP = 0x08; /* Write data to register */
    A1 = 0xFF; /* Write data to Port */
    status = P0; /* Read data from Port */
    EI = 1; /* Set a bit (enable all interrupts) */
}
```

To analyze this type of code, use these options:

- `Compiler` (`-compiler`): Specify `keil` or `iar`.
- `Sfr type support` (`-sfr-types`): Specify the data type and size in bits.

For example, depending on how you define the `sbit` data type, you use these options:

- `sbit ADST = ADCUP^7;`
Use options: `-compiler keil -sfr-type sfr=8`
- `sbit ADST = ADCUP.7;`
Use options: `-compiler iar -sfr-type sfr=8`

The analysis then supports the Keil or IAR language extensions even if some structures, keywords, and syntax are not part of the C99 standard.

Keywords Removed During Preprocessing

Once you specify the Keil or IAR compiler, the analysis recognizes compiler-specific keywords in your code. If a keyword is not relevant for the analysis, it is removed from the source code during preprocessing.

If you disable the keyword and use it as an identifier instead, you can encounter a compilation error when you compile your code with Polyspace. See “Fix Polyspace Compilation Errors Related to Keil or IAR Compiler” on page 32-34.

These keywords are removed during preprocessing:

- Keil: `bdata`, `far`, `idata`, `huge`, `sdata`
- IAR: `saddr`, `reentrant`, `reentrant_idata`, `non_banked`, `plm`, `bdata`, `idata`, `pdata`, `code`, `xdata`, `xhuge`, `interrupt`, `__interrupt`, `__intrinsic`

The `data` keyword is not removed.

Remove or Replace Keywords Before Compilation

The Polyspace compiler strictly follows the ANSI® C99 Standard (ISO/IEC 9899:1999). If your compiler allows deviation from the Standard, the Polyspace compilation using default options cannot emulate your compiler. For instance, your compiler can allow certain non-ANSI keyword, which Polyspace does not recognize by default.

To emulate your compiler closely, you specify the Target & Compiler options. If you still get compilation errors from unrecognized keywords, you can remove or replace them only for the purposes of verification. The option `Preprocessor definitions (-D)` allows you to make simple substitutions. For complex substitutions, for instance to remove a group of space-separated keywords such as a function attribute, use the option `Command/script` to apply to preprocessed files (`-post-preprocessing-command`).

Remove Unrecognized Keywords

You can remove unsupported keywords from your code for the purposes of analysis. For instance, follow these steps to remove the `far` and `0x` keyword from your code (`0x` precedes an absolute address).

- 1 Save the following template as `C:\Polyspace\myTpl.pl`.

Content of myTpl.pl

```
#!/usr/bin/perl

#####
# Post Processing template script
#
#####
# Usage from GUI:
#
# 1) Linux: /usr/bin/perl PostProcessingTemplate.pl
# 2) Windows: polyspaceroot\sys\perl\win32\bin\perl.exe <pathtoscript>\
# PostProcessingTemplate.pl
#
#####

$version = 0.1;

$INFILE = STDIN;
$OUTFILE = STDOUT;

while (<$INFILE>)
{
    # Remove far keyword
    s/far//;

    # Remove "@ 0xFE1" address constructs
    s/\@ \s0x[A-F0-9]*//g;

    # Remove "@0xFE1" address constructs
    s/\@0x[A-F0-9]*//g;

    # Remove "@ ((unsigned)&LATD*8)+2" type constructs
    s/\@ \s\(\(unsigned\) \&[A-Z0-9]+\*8\) \+ \d//g;


    # DON'T DELETE LINE BELOW: Print the current processed line
    print $OUTFILE $_;
}
```

For reference, see a summary of Perl regular expressions.

Perl Regular Expressions

```
#####
# Metacharacter What it matches
```

```
#####
# Single Characters
# . Any character except newline
# [a-z0-9] Any single character in the set
# [^a-z0-9] Any character not in set
# \d A digit same as
# \D A non digit same as [^0-9]
# \w An Alphanumeric (word) character
# \W Non Alphanumeric (non-word) character
#
# Whitespace Characters
# \s Whitespace character
# \S Non-whitespace character
# \n newline
# \r return
# \t tab
# \f formfeed
# \b backspace
#
# Anchored Characters
# \B word boundary when no inside []
# \B non-word boundary
# ^ Matches to beginning of line
# $ Matches to end of line
#
# Repeated Characters
# x? 0 or 1 occurrence of x
# x* 0 or more x's
# x+ 1 or more x's
# x{m,n} Matches at least m x's and no more than n x's
# abc All of abc respectively
# to|be|great One of "to", "be" or "great"
#
# Remembered Characters
# (string) Used for back referencing see below
# \1 or $1 First set of parentheses
# \2 or $2 First second of parentheses
# \3 or $3 First third of parentheses
#####
# Back referencing
#
# e.g. swap first two words around on a line
# red cat -> cat red
# s/(\w+) (\w+)/$2 $1/;
#
#####
```

- 2 On the **Configuration** pane, select **Environment Settings**.
- 3 To the right of **Command/script to apply to preprocessed files**, click .
- 4 Use the Open File dialog box to navigate to C:\Polyspace.
- 5 In the **File name** field, enter myTpl.pl.
- 6 Click **Open**. You see C:\Polyspace\myTpl.pl in the **Command/script to apply to preprocessed files** field.

Remove Unrecognized Function Attributes

You can remove unsupported function attributes from your code for the purposes of analysis.

If you run verification on this code specifying a generic compiler, you can see compilation errors from the `noreturn` attribute. The code compiles using a GNU compiler.

```
void fatal () __attribute__ ((noreturn));

void fatal (/* ... */)
{
    /* ... */ /* Print error message. */ /* ... */
    exit (1);
}
```

If the software does not recognize an attribute and the attribute does not affect the code analysis, you can remove it from your code for the purposes of verification. For instance, you can use this Perl script to remove the `noreturn` attribute.

```
while ($line = <STDIN>)
{
    # __attribute__ ((noreturn))

    # Remove far keyword
    $line =~ s/__attribute__ \(\(noreturn\)\)//g;

    # Print the current processed line to STDOUT
    print $line;
}
```

Specify the script using the option `Command/script` to apply to preprocessed files (`-post-preprocessing-command`).

See Also

Polyspace Analysis Options

`Command/script` to apply to preprocessed files (`-post-preprocessing-command`) |
Preprocessor definitions (`-D`)

Related Examples

- “Troubleshoot Compilation Errors”

Gather Compilation Options Efficiently

Polyspace verification can sometimes stop in the compilation or linking phase due to the following reasons:

- The Polyspace compiler strictly follows a C or C++ Standard (depending on your choice of compiler). See “C/C++ Language Standard Used in Polyspace Analysis” on page 13-5. If your compiler allows deviation from the Standard, the Polyspace compilation using default options cannot emulate your compiler.
- Your compiler declares standard library functions with argument or return types different from the standard types. Unless you also provide the function definition, for efficient verification, Polyspace uses its own definitions of standard library functions, which have the usual prototype. The mismatch in types causes a linking error.

You can easily work around the compilation and standard library function errors. To work around the errors, you typically specify certain analysis options. In some cases, you might have to add a few lines to your code. For instance:

- To emulate your compiler behavior more closely, you specify the Target & Compiler options. If you still face compilation errors, you might have to remove or replace certain unrecognized keywords using the option `Preprocessor definitions (-D)`. However, the option allows only simple substitution of a string with another string. For more complex replacements, you might have to add `#define` statements to your code.
- To avoid errors from stubbing standard library functions, you might have to `#define` certain Polyspace-specific macros so that Polyspace does not use its own definition of standard library functions.

Instead of adding these modifications to your original code, create a single `polyspace.h` file that contains all modifications. Use the option `Include (-include)` to force inclusion of the `polyspace.h` file in all source files under verification.

Benefits of this approach include:

- The error detection is much faster since it will be detected during compilation rather than in the link or subsequent phases.
- There will be no need to modify original source files.
- The file is automatically included as the very first file in the original `.c` files.
- The file is reusable for other projects developed under the same environment.

Example 13.1. Example

This is an example of a file that can be used with the option `Include (-include)`.

```
// The file may include (say) a standard include file implicitly
// included by the cross compiler

#include <stdlib.h>
#include "another_file.h"

// Workarounds for compilation errors
#define far
#define at(x)
```

```
// Workarounds for errors due to redefining standard library functions

#define POLYSPACE_NO_STANDARD_STUBS // use this flag to prevent the
    //automatic stubbing of std functions
#define __polyspace_no_sscanf
#define __polyspace_no_fgetc
void sscanf(int, char, char, char, char, char);
void fgetc(void);
```

See Also

More About

- “Troubleshoot Compilation Errors”

Configure Inputs and Stubbing Options

Specify External Constraints for Polyspace Analysis

Polyspace products analyzes C/C++ code and checks for issues such as defects (bugs) or run-time errors. The analysis uses the code that you provide to make assumptions about items such as variable ranges and allowed buffer size for pointers. Sometimes the assumptions are broader than what you expect because:

- You have not provided the complete code. For example, you did not provide some of the function definitions.
- Some of the information about variables is available only at run time. For example, some variables in your code obtain values from the user at run time.

Because of these broad assumptions:

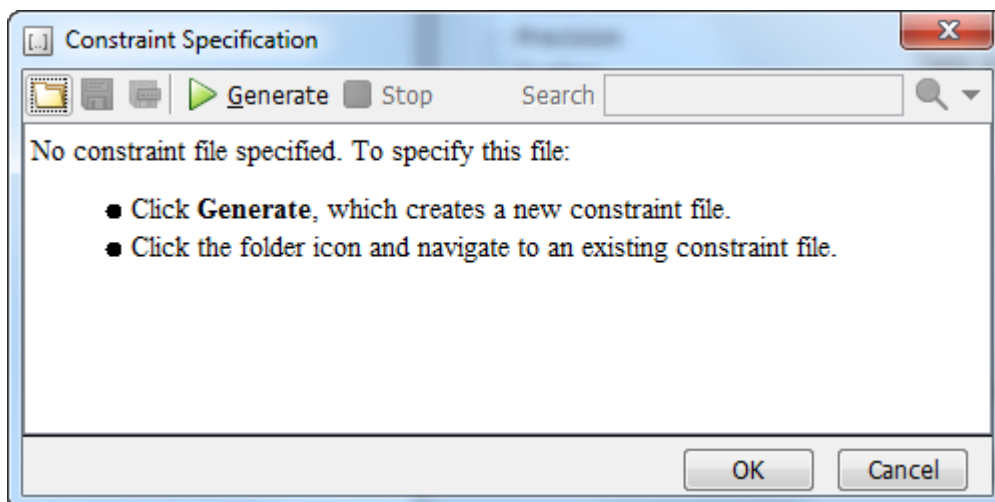
- Code Prover can consider more execution paths than those paths that occur at run time. If an operation fails along one of the execution paths, Polyspace places an orange check on the operation. If that execution path comes from an assumption that is too broad, the orange check might indicate a false positive.
- Bug Finder can sometimes produce false positives.

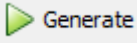
To reduce the number of such false positives, you can specify additional constraints on global variables, function inputs, and return values and modifiable arguments of stubbed functions. This example shows how to specify these external constraints (also known as data range specifications or DRS). You save the constraints as an XML file to use them for subsequent analyses. If your source code changes, you can update the previous constraints. You do not have to create a new constraint template.

Create Constraint Template

User Interface (Desktop Products Only)

- 1 Open the project configuration. On the **Configuration** pane, select **Inputs & Stubbing**.
- 2 To the right of **Constraint setup**, click the **Edit** button to open the **Constraint Specification** window.



- 3 In the Constraint Specification dialog box, create a blank constraint template. The template contains a list of all variables on which you can provide constraints. To create a new template, click . The software compiles your project and creates a template. The new template is stored in a file `Module_number_Project_name_drs_template.xml` in your project folder.
- 4 Specify your constraints and save the template as an XML file. For more information, see “External Constraints for Polyspace Analysis” on page 14-6.
- 5 Click **OK**.

You see the full path to the template XML file in the **Constraint setup** field. If you run an analysis, Polyspace uses this template for extracting variable constraints.

Command Line

Use the option `Constraint setup (-data-range-specifications)` to specify the constraints XML file.

To specify constraints in the XML file:

- 1 First, create a blank XML template. The template lists all global variables, function inputs and modifiable arguments and return values of stubbed functions without specifying any constraints on them.

To create a blank template, run an analysis only up to the compilation phase. In Bug Finder, disable checking of defects. Use the option `Find defects (-checkers)`. In Code Prover, check for source compliance only. Use the argument `compile` for the option `Verification level (-to)`. After the analysis, a blank template XML `drs-template.xml` is created in the results folder.

For C++ projects, to create a blank constraints template, you have to use the argument `cpp-normalize` for the option `Verification level (-to)`.

- 2 Edit the XML file to specify your constraints.

For examples, see:


- “Constrain Global Variable Range for Polyspace Analysis” on page 14-12
- “Constrain Function Inputs for Polyspace Analysis” on page 14-14

Create Constraint Template from Code Prover Analysis Results

You can constrain variable ranges based on their expected range in real-world applications. For instance, if a variable represents vehicle speed, you can set a maximum possible value. You can also constrain variable ranges only if they cause too many orange checks from overapproximation.

A Code Prover analysis shows all global variables, function inputs and stubbed functions that lead to orange checks from possible overapproximation. You can constrain only these variables for a more precise analysis.

- 1 Open Code Prover results in the Polyspace user interface or Polyspace Access web interface.
- 2 Open the **Orange Sources** pane. Do one of the following:

- Select an orange check. If the software can trace an orange check to a root cause, a  icon appears on the **Result Details** pane. Click this icon to open the **Orange Sources** pane.
- In the Polyspace user interface, select **Window > Show/Hide View > Orange Sources**. In the Polyspace Access web interface, select **Window > Orange Sources**.

You see the full list of variables (function inputs or return values of stubbed functions) that can cause orange checks. Constrain the ranges of these variables.

In the details for individual orange checks, you often see a message similar to this:

If appropriate, applying DRS to stubbed function `random_float` in `example.c` line 44 may remove this orange.


The message is an indication that the stubbed function is a possible source of the orange check. You can apply external constraints on the function to enforce more precise assumptions and possibly remove the orange check (in case it came from the broader assumptions).

Update Existing Template

With new code submissions, you might have to specify additional constraints. You can update an existing template to add global variables, function inputs and stubbed functions that come from the new code submissions.

Additionally, if you remove some variables or functions from your code, constraints on them are not applicable any more. Instead of regenerating a constraint template and respecifying the constraints, you can update an existing template and remove the variables that are not present in your code.

User Interface (Desktop Products Only)

- 1 On the **Configuration** pane, select **Inputs & Stubbing**.
- 2 Open the existing template in one of the following ways:
 - In the **Constraint setup** field, enter the path to the template XML file. Click **Edit**.
 - Click **Edit**. In the Constraint Specification dialog box, click the  icon to navigate to your template file.
- 3 Click **Update**.
 - a Variables that are no longer present in your source code appear under the **Non Applicable** node. To remove an entry under the **Non Applicable** node or the node itself, right-click and select **Remove This Node**.
 - b Specify your new constraints for any of the other variables.

Command Line

In a continuous integration workflow, you can use the constraints XML file from the previous run. If new code submissions require additional constraints:

- 1 Specify constraints on variables from new code submissions in a constraints XML file. See [Create Constraint Template: Command Line](#) on page 14-3.
- 2 Merge the constraints XML file with the new constraints and the constraints XML file from the previous run.

Specify Constraints in Code

Specifying constraints outside your code allows for more precise analysis. However, you must use the code within the specified constraints because the constraints are *outside* your code. Otherwise, the results might not apply. For example, if you use function inputs outside your specified range, a run-time error can occur on an operation even though checks on the operation are green.

To specify constraints *inside* your code, you can use:

- Appropriate error handling tests in your code.

Polyspace checks to determine if the errors can actually occur. If they do not occur, the test blocks appear as **Unreachable code**.

- The `unchecked_assert` macro. For example, to constrain a variable `var` in the range `[0,10]`, you can use `assert(var >= 0 && var <=10);`.

See “Constrain Variable Ranges for Polyspace Analysis Using Manual Stubs and Manual main() Function” (Polyspace Code Prover).

See Also

Constraint setup (-data-range-specifications)

Related Examples

- “External Constraints for Polyspace Analysis” on page 14-6
- “Constrain Global Variable Range for Polyspace Analysis” on page 14-12
- “Constrain Function Inputs for Polyspace Analysis” on page 14-14
- “XML File Format for Polyspace Analysis Constraints” on page 14-17

External Constraints for Polyspace Analysis

Polyspace uses the code that you provide to make assumptions about items such as variable ranges and allowed buffer size for pointers. Sometimes the assumptions that Polyspace makes are broader than what you expect, which might result in Bug Finder false positives or more Code Prover orange checks. To reduce such false positives or orange checks, you can specify external constraints on:

- Global variables.
- User-defined functions.
- Stubbed functions.

For more information, see “Specify External Constraints for Polyspace Analysis” on page 14-2. For a partial list of limitations, see “Constraint Specification Limitations” on page 14-11.

Effect of External Constraints

Consider the following function:

```
int getFlooredNumber(int total, int size) {
    return total/size;
}
```

Since the input `size` is unknown, if you analyze this function:

- With Polyspace Code Prover, you see an orange **Division by zero** check. The orange check indicates that Code Prover suspects a possible division by zero error, but this error does not occur on all execution paths.
- With Polyspace Bug Finder, if you use the option `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`, you see an **Integer division by zero** defect along with one possible input value leading to the defect (in this case, a size of 0).

In more sophisticated examples, for instance, if the division occurs inside a condition, a defect appears from unknown inputs even without using the option.

In both cases, the analysis determines possible values of the input variable `size` from its data type. Since the variable `size` has data type `int`, on targets where `int` has a size of 32 bits, the variable is assumed to have values in the range $[-2^{31}, 2^{31}-1]$.

If you know that an input has values only within a certain range, you can specify an external constraint on the input (also known as Data Range Specification or DRS). For instance, in the above example, if you specify a range on `size` that excludes zero:

- Code Prover no longer shows the orange **Division by zero** check.
- Bug Finder no longer shows the **Integer division by zero** defect.

You can specify external constraints to emulate design constraints that live outside your code. For instance, if an input variable represents a physical quantity such as vehicle speed, you can constrain the variable values to speeds for which the vehicle is designed.

Constraint Specification

In the user interface of the Polyspace desktop products, you can specify the constraints through a **Constraint Specification** window. The constraints are saved in an XML file that you can reuse for other projects.

Name	File	Attributes	Data ...	Main Generator Called	Init Mode	Init Range	Initialize Pointer	Init Allocated
Global Variables								
User Defined Functions								
isEndOfList()	file.c	unused		MAIN GENERATOR				
isEndOfList.arg1	file.c		int32		INIT	-1..0		
isEndOfList.arg2	file.c		int32 *		MAIN GENERATOR		Not NULL	MAIN GENERATOR
isEndOfList.* arg	file.c		int32		MAIN GENERATOR			
isEndOfList.return	file.c		int32					
Stubbed Functions								
Non Applicable								

This table describes the various columns in the **Constraint Specification** window. If you directly edit the constraint XML file to specify a constraint (for instance, in the Polyspace Server products), this table also shows the correspondence between columns in the user interface and entries in the XML file. The XML entry highlighted in bold appears in the corresponding column of the **Constraint Specification** window.

Column	Settings
Name	<p>Displays the list of variables and functions in your Project for which you can specify data ranges.</p> <p>This Column displays three expandable menu items:</p> <ul style="list-style-type: none"> • Globals - Displays global variables in the project. • User defined functions - Displays user-defined functions in the project. Expand a function name to see its inputs. • Stubbed functions - Displays a list of stub functions in the project. Expand a function name to see the inputs and return values. <p>XML File Entry:</p> <pre><function name = "funcName" ...> <scalar name = "arg1" ...> <pointer name = "arg2" ...></pre>
File	<p>Displays the name of the source file containing the variable or function.</p> <p>XML File Entry:</p> <pre><file name = "C:\Project1\Sources\file.c" ...></pre>
Attributes	<p>Displays information about the variable or function.</p> <p>For example, static variables display <code>static</code>. Uncalled functions display <code>unused</code>.</p>

Column	Settings
	<p>XML File Entry:</p> <pre><function name="funcName" attributes="unused" ...></pre>
Data Type	<p>Displays the variable type.</p> <p>XML File Entry:</p> <pre><scalar name="arg1" complete_type="int32" ...></pre>
Main Generator Called	<p>Applicable only for user-defined functions.</p> <p>Specifies whether the main generator calls the function:</p> <ul style="list-style-type: none"> • MAIN GENERATOR - Main generator may call this function, depending on the value of the <code>-functions-called-in-loop (C)</code> or <code>-main-generator-calls (C++)</code> parameter. • NO - Main generator will not call this function. • YES - Main generator will call this function. <p>XML File Entry:</p> <pre><function name="funcName" main_generator_called="MAIN_GENERATOR" ...></pre>
Init Mode	<p>Specifies how the software assigns a range to the variable:</p> <ul style="list-style-type: none"> • MAIN GENERATOR - Variable range is assigned depending on the settings of the main generator options <code>-main-generator-writes-variables</code> and <code>-no-def-init-glob</code>. • IGNORE - Variable is not assigned to any range, even if a range is specified. • INIT - Variable is assigned to the specified range only at initialization, and keeps the range until first write. • PERMANENT - Variable is permanently assigned to the specified range. If the variable is assigned outside this range during the program, no warning is provided. Use the <code>globalassert</code> mode if you need a warning. <p>User-defined functions support only INIT mode.</p> <p>Stub functions support only PERMANENT mode.</p> <p>For C verifications, global pointers support MAIN GENERATOR, IGNORE, or INIT mode.</p> <ul style="list-style-type: none"> • MAIN GENERATOR - Pointer follows the options of the main generator. • IGNORE - Pointer is not initialized • INIT - Specify if the pointer is NULL, and how the pointed object is allocated (Initialize Pointer and Init Allocated options). <p>XML File Entry:</p> <pre><scalar name="arg1" init_mode="INIT" ...></pre>

Column	Settings
Init Range	<p>Specifies the minimum and maximum values for the variable.</p> <p>You can use the keywords <code>min</code> and <code>max</code> to denote the minimum and maximum values of the variable type. For example, for the type <code>long</code>, <code>min</code> and <code>max</code> correspond to -2^{31} and $2^{31}-1$ respectively.</p> <p>You can also use hexadecimal values. For example: <code>0x12..0x100</code></p> <p>For <code>enum</code> variables, you cannot specify ranges directly using the enumerator constants. Instead use the values represented by the constants.</p> <p>For <code>enum</code> variables, you can also use the keywords <code>enum_min</code> and <code>enum_max</code> to denote the minimum and maximum values that the variable can take. For example, for an <code>enum</code> variable of the type defined below, <code>enum_min</code> is 0 and <code>enum_max</code> is 5:</p> <pre>enum week{ sunday, monday=0, tuesday, wednesday, thursday, friday, saturday};</pre> <p>XML File Entry:</p> <pre><scalar name="arg1" init_range="-1..0"...></pre>
Initialize Pointer	<p>Applicable only to pointers. Enabled only when you specify Init Mode:INIT.</p> <p>Specifies whether the pointer should be NULL:</p> <ul style="list-style-type: none"> • May-be NULL - The pointer could potentially be a NULL pointer (or not). • Not Null - The pointer is never initialized as a null pointer. • Null - The pointer is initialized as NULL. <p>Note Not applicable for C++ projects. See “Constraint Specification Limitations” on page 14-11.</p> <p>XML File Entry:</p> <pre><pointer name="arg1" initialize_pointer="Not NULL"...></pre>
Init Allocated	<p>Applicable only to pointers. Enabled only when you specify Init Mode:INIT.</p> <p>Specifies how the pointed object is allocated:</p> <ul style="list-style-type: none"> • MAIN GENERATOR - The pointed object is allocated by the main generator. • None - Pointed object is not written. • SINGLE - Write the pointed object or the first element of an array. (This setting is useful for stubbed function parameters.) • MULTI - All objects (or array elements) are initialized. <p>Note Not applicable for C++ projects. See “Constraint Specification Limitations” on page 14-11.</p>

Column	Settings
	<p>XML File Entry:</p> <pre data-bbox="516 352 1295 382"><pointer name="arg1" init_pointed="MAIN_GENERATOR"...></pre>
<p># Allocated Objects</p>	<p>Applicable only to pointers.</p> <p>Specifies how many objects are pointed to by the pointer (the pointed object is considered as an array).</p> <p>The Init Allocated parameter specifies how many allocated objects are actually initialized. For instance, consider this code:</p> <pre data-bbox="516 632 829 741">void func(int *ptr) { assert(ptr[0]==1); assert(ptr[1]==1); }</pre> <p>If you specify these constraints:</p> <ul data-bbox="516 825 1435 898" style="list-style-type: none"> • ptr has Init Allocated set to MULTI and # Allocated Objects set to 2, • *ptr has Init Range set to 1..1, <p>both assertions are green. However, if you specify these constraints:</p> <ul data-bbox="516 978 1024 1052" style="list-style-type: none"> • ptr has Init Allocated set to SINGLE • *ptr has Init Range set to 1..1, <p>the second assertion is orange. Only the first object that ptr points to initialized to 1. Objects beyond the first can be potentially full range.</p> <p>Use the keyword "max" to specify that a pointer can point to anywhere within a very large array of unknown file size. In your analysis results, you see very large offsets and buffer sizes for the pointer. The offset and buffer sizes depend on the pointer size and other characteristics of the target processor type used. The largest object Polyspace creates using this method is a buffer of 2²⁷-1 bytes (134217726 bytes).</p> <hr/> <p>Note Not applicable for C++ projects. See "Constraint Specification Limitations" on page 14-11.</p> <hr/> <p>XML File Entry:</p> <pre data-bbox="516 1545 1179 1575"><pointer name="arg1" number_allocated="10"...></pre>
<p>Global Assert</p>	<p>Specifies whether to perform an assert check on the variable at global initialization, and after each assignment.</p> <p>XML File Entry:</p> <pre data-bbox="516 1713 1135 1743"><scalar name="glob" global_assert="YES"...></pre>
<p>Global Assert Range</p>	<p>Specifies the minimum and maximum values for the range you want to check.</p> <p>XML File Entry:</p> <pre data-bbox="516 1854 1166 1883"><scalar name="glob" assert_range="0..200"...></pre>

Column	Settings
Comment	Remarks that you enter, for example, justification for your DRS values.
	XML File Entry: <pre><scalar name="glob" comment="Speed Range" ...></pre>

Constraint Specification Limitations

You cannot specify the following types of constraints using the constraint specification interface. To work around some of these limitations, you can define your own stubs and `main()`. For details, see “Constrain Variable Ranges for Polyspace Analysis Using Manual Stubs and Manual `main()` Function” (Polyspace Code Prover).

The constraint specification interface does not support these kinds of constraints:

- In C++, you cannot constrain pointer or reference arguments of functions.

Because of polymorphism, a C++ pointer or reference can point to objects of multiple classes in a class hierarchy and can require invoking different constructors. The pre-analysis for constraint specification cannot determine which object type to constrain or which constructor to call.

- You cannot specify a constraint that relates the return value of a function to its inputs. You can specify only a constant range for the constraints.
- You cannot specify multiple ranges for a constraint. For instance, you cannot specify that a function argument has either the value -1 or a value in the range [1,100]. Instead, specify the range [-1,100] or perform two separate analyses, once with the value -1 and once with the range [1,100].
- You cannot specify separate constraints on different fields of a union.

See Also

More About


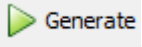
- “Specify External Constraints for Polyspace Analysis” on page 14-2

Constrain Global Variable Range for Polyspace Analysis

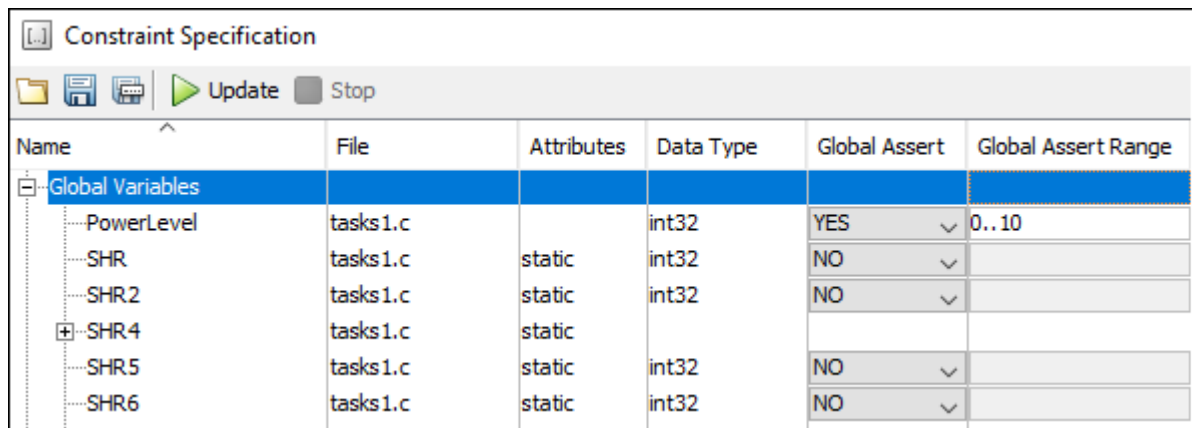
You can impose external constraints (also known as data range specifications or DRS) on the range of global variables in C/C++ code and check with Polyspace Code Prover whether write operations on the variable violate the constraint. For the general workflow, see “Specify External Constraints for Polyspace Analysis” on page 14-2.

User Interface (Desktop Products Only)


To constrain a global variable range and also check for violation of the constraint:

- 1 In your project configuration, select **Inputs & Stubbing**. Click the  button next to the **Constraint setup** field.
- 2 In the Constraint Specification window, click .

Under the **Global Variables** node, you see a list of global variables.



Name	File	Attributes	Data Type	Global Assert	Global Assert Range
Global Variables					
PowerLevel	tasks1.c		int32	YES	0..10
SHR	tasks1.c	static	int32	NO	
SHR2	tasks1.c	static	int32	NO	
SHR4	tasks1.c	static			
SHR5	tasks1.c	static	int32	NO	
SHR6	tasks1.c	static	int32	NO	

- 3 For the global variable that you want to constrain:
 - From the drop-down list in the **Global Assert** column, select YES.
 - In the **Global Assert Range** column, enter the range in the format *min*..*max*. *min* is the minimum value and *max* the maximum value for the global variable.
 - 4 To save your specifications, click the  button.
- In **Save a Constraint File** window, save your entries as an xml file.
- 5 Run a verification and open the results.

For every write operation on the global variable, you see a green, orange, or red **Correctness condition** check. If the check is:

- Green, the variable is within the range that you specified.
- Orange, the variable can be outside the range that you specified.
- Red, the variable is outside the range that you specified.

When two or more tasks write to the same global variable, the **Correctness condition** check can appear orange on all write operations to the variable even when only one write operation takes the variable outside the **Global Assert** range.

Command Line

Use the option `Constraint setup (-data-range-specifications)` with an XML file specifying your constraint.

For instance, for an analysis with Polyspace Code Prover Server, specify the option as follows:

```
polyspace-code-prover-server -sources filename
                             -data-range-specifications "C:\Polyspace\drs_project1.xml"
```

Create a blank constraint XML template as described in “Specify External Constraints for Polyspace Analysis” on page 14-2. In the XML file, locate and constrain the global variables. XML tags for global variables appear directly within the `file` tag without an enclosing function tag. For instance, in this constraint XML, `PowerLevel` and `SHR` are global variables:

```
<file name="\\\\home\\Polyspace_Workspace\\Examples\\Code_Prover_Example
           \\sources\\tasks1.c">
  <scalar name="PowerLevel" line="26" .. global_assert="YES" assert_range="0..10"/>
  <scalar name="SHR" line="30" ... global_assert="NO" assert_range="" />
  <function name="Tserver" line="73" .../>
  <function name="initregulate" line="47" .../>
  <function name="orderregulate" line="35" ...>
    <scalar name="return" ... global_assert="unsupported" assert_range="unsupported" />
  </function>
  <function name="procl" line="101" .../>
</file>
```

To specify a constraint on a global variable and check during a Code Prover analysis if the constraint is violated:

- 1 Set the `global_assert` attribute of the variable's `scalar` tag to `YES`.
- 2 Set the `assert_range` attribute to a range in the form `min..max`, for instance, `0..10`.

In the preceding example, the variable `PowerLevel` is constrained this way.

See Also

Polyspace Analysis Options

`Constraint setup (-data-range-specifications)`

Polyspace Results

Correctness condition

More About

- “Specify External Constraints for Polyspace Analysis” on page 14-2
- “External Constraints for Polyspace Analysis” on page 14-6
- “Constrain Function Inputs for Polyspace Analysis” on page 14-14

Constrain Function Inputs for Polyspace Analysis

If a program module analyzed with Polyspace Code Prover does not contain a `main` function, the analysis by default starts effectively from all uncalled functions¹. Since these functions are not called within the code, Code Prover has to make assumptions about the function inputs based on their data types. For a more precise Code Prover analysis, you can specify constraints (also known as data range specifications or DRS) on these function inputs. Code Prover analyzes these functions for run-time errors with respect to the constrained inputs. For the general workflow, see “Specify External Constraints for Polyspace Analysis” on page 14-2.

For instance, for a function defined as follows, you can specify that the argument `val` has values in the range `[1..10]`. You can also specify that the argument `ptr` points to a 3-element array where each element is initialized:

```
int func(int val, int* ptr) {
    .
    .
}
```

A similar assumption about function inputs is seen in Bug Finder if you use the option `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`. You can also constrain a Bug Finder analysis with external constraints.

Note that if a function is called within the code, the external constraints no longer apply. Code Prover tracks the data flow within the code and analyzes a called function with actual arguments used in the code.

User Interface (Desktop Products Only)

To specify constraints on function inputs:

- 1 In your project configuration, select **Inputs & Stubbing**. Click the  button for **Constraint setup**.

- 2 In the Constraint Specification window, click .

Under the **User Defined Functions** node, you see a list of functions whose inputs can be constrained.

- 3 Expand the node for each function.

You see each function input on a separate row. The inputs have the syntax `function_name.arg1`, `function_name.arg2`, etc.

- 4 Specify your constraints on one or more of the function inputs. For more information, see “External Constraints for Polyspace Analysis” on page 14-6.

For example, in the preceding code:

- To constrain `val` to the range `[1..10]`, select `INIT` for **Init Mode** and enter `1..10` for **Init Range**.

¹ The Code Prover analysis generates a `main` that calls all uncalled functions by default and starts analysis from this `main`. You can change this default behavior using the option `Functions to call (-main-generator-calls)`.

- To specify that `ptr` points to a 3-element array where each element is initialized, select **MULTI** for **Init Allocated** and enter 3 for **# Allocated Objects**.

Name	File	Attributes	Data Type	Main Generator Called	Init Mode	Init Range	Initialize Pointer	Init Allocated	# Allocated Objects
Global Variables									
User Defined Functions									
func()	file.c	unused		MAIN GENERATOR					
func.arg1	file.c		int32		INIT	1..10			
func.arg2	file.c		int32 *		INIT		Not NULL	MULTI	3
func.* arg2	file.c		int32		MAIN GENE...				
func.return	file.c		int32						

- Run verification and open the results. On the **Source** pane, place your cursor on the function inputs.

The tooltips display the constraints. For example, in the preceding code, the tooltip displays that `val` has values in `1..10`.

Command Line

Use the option `Constraint setup (-data-range-specifications)` with an XML file specifying your constraint.

For instance, for an analysis with Polyspace Code Prover Server, specify the option as follows:

```
polyspace-code-prover-server -sources filename
                             -data-range-specifications "C:\Polyspace\drs_project1.xml"
```

Create a blank constraint XML template as described in “Specify External Constraints for Polyspace Analysis” on page 14-2. In the XML file, locate and constrain the function inputs. The function inputs appear as a `scalar` or `pointer` tag in a `function` tag. The inputs are named as `arg1`, `arg2` and so on. For instance, for the preceding code, the XML structure for the inputs of `func` appear as follows:

```
<function name="func" line="1" attributes="unused"
  main_generator_called="MAIN_GENERATOR" comment="">
  <scalar name="arg1" line="1" base_type="int32"
    complete_type="int32" init_mode="INIT" init_range="1..10" />
  <pointer name="arg2" line="1" complete_type="int32 *"
    init_mode="INIT" initialize_pointer="Not NULL" number_allocated="3"
    init_pointed="MULTI">
    <scalar line="1" base_type="int32" complete_type="int32"
      init_mode="MAIN_GENERATOR" init_range="" />
  </pointer>
  <scalar name="return" line="1" base_type="int32" complete_type="int32"
    init_mode="disabled" init_range="disabled" />
</function>
```

To specify a constraint on a function input, set the attributes `init_mode` and `init_range` for scalar variables, and `init_pointed` and `number_allocated` for pointer variables.

- To constrain `val` to the range `[1..10]`, set the `init_mode` attribute of the tag with name `arg1` to `INIT` and `init_range` to `1..10`.

- To specify that `ptr` points to a 3-element array where each element is initialized, set the `init_mode` attribute of the tag with name `arg2` to `INIT`, `init_pointed` to `MULTI` and `number_allocated` to 3.

See Also

Constraint setup (-data-range-specifications)

More About

- “Specify External Constraints for Polyspace Analysis” on page 14-2
- “External Constraints for Polyspace Analysis” on page 14-6
- “Constrain Global Variable Range for Polyspace Analysis” on page 14-12

XML File Format for Polyspace Analysis Constraints

For a more precise Polyspace analysis, you can specify constraints on global variables, function inputs and stubbed functions. You can specify the constraints in the user interface of the Polyspace desktop products or at the command line as an XML file. For the general workflow, see “Specify External Constraints for Polyspace Analysis” on page 14-2.

This topic describes details of the constraint XML file schema. You typically require this information only if you create a constraint XML from scratch. If you run a verification once, the software automatically generates a template constraint file `drs-template.xml` in your results folder. Instead of creating a constraint XML file from scratch, it is easier to edit this template XML file to specify your constraints. For some examples, see:

- “Constrain Global Variable Range for Polyspace Analysis” on page 14-12
- “Constrain Function Inputs for Polyspace Analysis” on page 14-14

For another explanation of what the XML tags mean, see “External Constraints for Polyspace Analysis” on page 14-6.

You can also see the information in this topic and the underlying XML schema in `polyspaceroot\polyspace\drs`. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

Syntax Description — XML Elements

The constraints file contains the following XML elements:

- `<global>` element — Declares the global scope, and is the root element of the XML file.
- `<file>` element — Declares a file scope. Must be enclosed in the `<global>` element. May enclose any variable or function declaration. Static variables must be enclosed in a file element to avoid conflicts.
- `<scalar>` element— Declares an integer or a floating point variable. May be enclosed in any recognized element, but cannot enclose any element. Sets `init/permanent/global` asserts on variables.
- `<pointer>` element — Declares a pointer variable. May enclose any other variable declarations (including itself), to define the pointed objects. Specifies what value is written into pointer (NULL or not), how many objects are allocated and how the pointed objects are initialized.
- `<array>` element — Declares an array variable. May enclose any other variable definition (including itself), to define the members of the array.
- `<struct>` element — Declares a structure variable or object (instance of class). May enclose any other variable definition (including itself), to define the fields of the structure.
- `<function>` element — Declares a function or class method scope. May enclose any variable definition, to define the arguments and the return value of the function. Arguments should be named `arg1`, `arg2`, ..., `argn` and the return value should be called `return`.

The following notes apply to specific fields in each XML element:

- **(*)** — Fields used only by the GUI. These fields are not mandatory for verification to accept the ranges. The field `line` contains the line number where the variable is declared in the source code, `complete_type` contains a string with the complete variable type, and `base_type` is used by the

GUI to compute the min and max values. The field comment is used to add information about any node.

- **(**)** — The field name is mandatory for scope elements `<file>` and `<function>` (except for function pointers). For other elements, the name must be specified when declaring a root symbol or a struct field.
- **(***)** — If more than one attribute applies to the variable, the attributes must be separated by a space. Only the static attribute is mandatory, to avoid conflicts between static variables having the same name. An attribute can be defined multiple times without impact.
- **(****)** — This element is used only by the GUI, to determine which `init` modes are allowed for the current element (according to its type). The value works as a mask, where the following values are added to specify which modes are allowed:
 - **1**: The mode “NO” is allowed.
 - **2** : The mode “INIT” is allowed.
 - **4**: The mode “PERMANENT” is allowed.
 - **8**: The mode “MAIN_GENERATOR” is allowed.

For example, the value “**10**” means that modes “INIT” and “MAIN_GENERATOR” are allowed. To see how this value is computed, refer to “Valid Modes and Default Values” on page 14-21.

- **(*****)** — A sub-element of a pointer (i.e. a pointed object) will be taken into account only if `init_pointed` is equal to `SINGLE`, `MULTI`, `SINGLE_CERTAIN_WRITE` or `MULTI_CERTAIN_WRITE`.
- **(*****)** — `SINGLE_CERTAIN_WRITE` or `MULTI_CERTAIN_WRITE` are available for parameters and return values of stubbed functions only if they are pointers. If the parameter or return value is a structure and the structure has a pointer field, they are also available for the pointer field.

<file> Element

Field	Syntax
name	<i>filepath_or_filename</i>
comment	<i>string</i>

<scalar> Element

Field	Syntax
name (**)	<i>name</i>
line (*)	<i>line</i>
base_type (*)	intx uintx floatx
Attributes (***)	volatile extern static const
complete_type (*)	<i>type</i>

Field	Syntax
init_mode	MAIN_GENERATOR IGNORE INIT PERMANENT disabled unsupported
init_modes_allowed (*)	<i>single value (****)</i>
init_range	<i>range</i> disabled unsupported
global_assert	YES NO disabled unsupported
assert_range	<i>range</i> disabled unsupported
comment(*)	<i>string</i>

<pointer> Element

Field	Syntax
Name (**)	<i>name</i>
line (*)	<i>line</i>
Attributes (***)	volatile extern static const
complete_type (*)	<i>type</i>
init_mode	MAIN_GENERATOR IGNORE INIT PERMANENT disabled unsupported
init_modes_allowed (*)	<i>single value (****)</i>
initialize_pointer	May be: NULL Not NULL NULL
number_allocated	<i>single value</i> disabled unsupported

Field	Syntax
init_pointed (*****)	MAIN_GENERATOR NONE SINGLE MULTI SINGLE_CERTAIN_WRITE MULTI_CERTAIN_WRITE disabled
comment	<i>string</i>

<array> and <struct> Elements

Field	Syntax
Name (**)	<i>name</i>
line (*)	<i>line</i>
complete_type (*)	<i>type</i>
attributes (***)	volatile extern static const
comment	<i>string</i>

<function> Element

Field	Syntax
Name (**)	<i>name</i>
line (*)	<i>line</i>
main_generator_called	MAIN_GENERATOR YES NO disabled
attributes (***)	static extern unused
comment	<i>string</i>

Valid Modes and Default Values

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
Global variables	Base type	Unqualified/static/const scalar	MAIN_GENERATOR IGNORE INIT	YES NO			Main generator dependent
			PERMANENT	disabled			
		Volatile scalar	PERMANENT	disabled			PERMANENT min..max
		Extern scalar	INIT	YES NO			INIT min..max
	PERMANENT		disabled				
	Struct	Struct field	Refer to field type				
Array	Array element	Refer to element type					
Global variables	Pointer	Unqualified/static/const scalar	MAIN_GENERATOR IGNORE INIT		May be NULL Not NULL NULL	NONE SINGLE MULTI	Main generator dependent
		Volatile pointer	un-supported		un-supported	un-supported	
		Extern pointer	IGNORE INIT		May be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May be NULL max MULTI
		Pointed volatile scalar	un-supported	un-supported			
		Pointed extern scalar	INIT	un-supported			INIT min..max
		Pointed other scalars	MAIN_GENERATOR INIT	un-supported			MAIN_GENERATOR dependent
		Pointed pointer	MAIN_GENERATOR INIT/	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	MAIN_GENERATOR dependent
		Pointed function	un-supported	un-supported			
Function parameters	Userdef function	Scalar parameters	MAIN_GENERATOR INIT	un-supported			INIT min..max

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default	
		Pointer parameters	MAIN_GENERATOR INIT	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May be NULL max MULTI	
		Other parameters	Refer to parameter type					
	Stubbed function	Scalar parameter	disabled	un-supported				
		Pointer parameters	disabled		disabled	NONE SINGLE MULTI SINGLE_CERTAIN_ WRITE MULTI_CERTAIN_ WRITE	MULTI	
		Pointed parameters	PERMANENT	un-supported			PERMANENT min..max	
		Pointed const parameters	disabled	un-supported				
Function return	Userdef function	Return	disabled	un-supported	disabled	disabled		
	Stubbed function	Scalar return	PERMANENT	un-supported			PERMANENT min..max	
		Pointer return	PERMANENT	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI SINGLE_CERTAIN_ WRITE MULTI_CERTAIN_ WRITE	PERMANENT May be NULL max MULTI	

See Also

More About

- “Specify External Constraints for Polyspace Analysis” on page 14-2
- “Constrain Global Variable Range for Polyspace Analysis” on page 14-12
- “Constrain Function Inputs for Polyspace Analysis” on page 14-14

Configure Multitasking Analysis

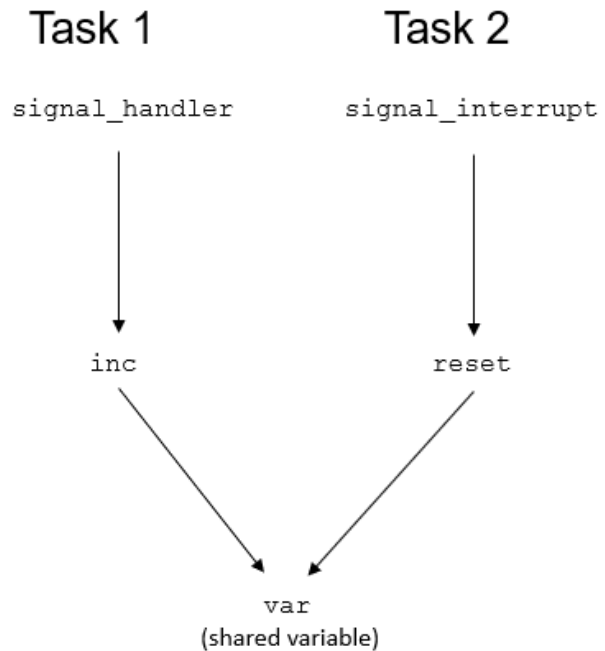
Analyze Multitasking Programs in Polyspace

With Polyspace, you can analyze programs where multiple threads (tasks) run concurrently.

```

1  int var;
2
3  void reset(void) {
4      var=0;
5  }
6
7  void inc(void) {
8      var+=2;
9  }
10
11  /* Task 1 */
12  void signal_handler(void) {
13      volatile int randomValue = 0;
14      while(randomValue) {
15          inc();
16      }
17  }
18
19  /* Task 2 */
20  void signal_interrupt(void) {
21      volatile int randomValue = 0;
22      while(randomValue) {
23          reset();
24      }
25  }
26
27  void main() {
28  }

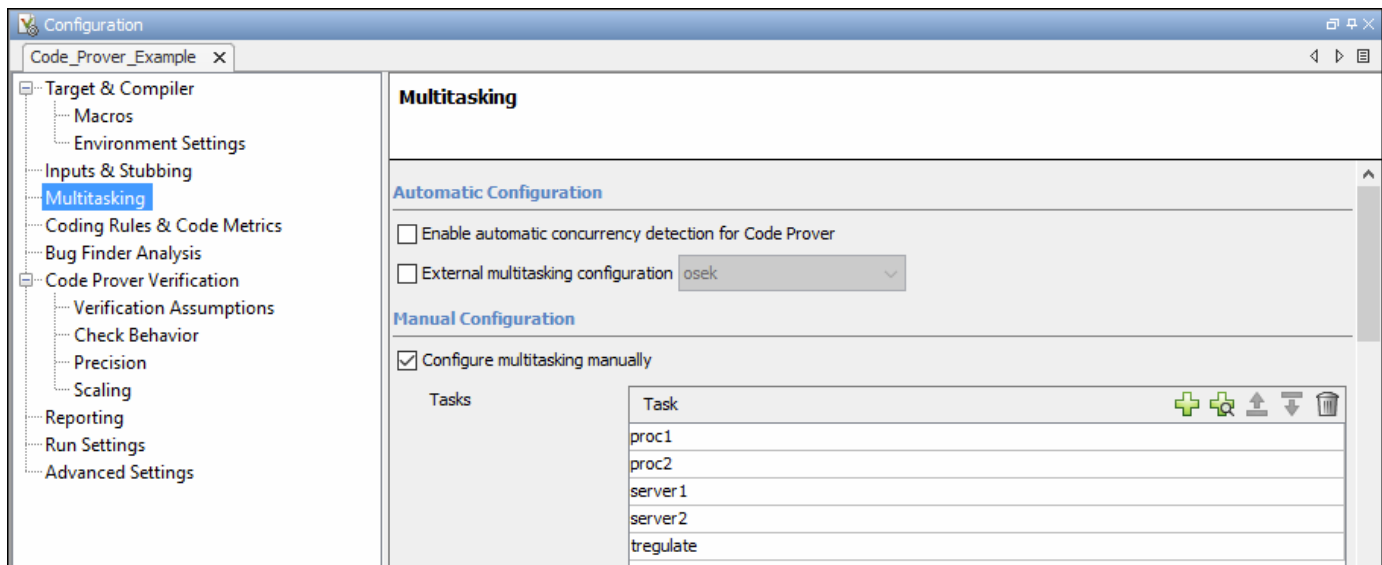
```



In addition to regular run-time checks, the analysis looks for issues specific to concurrent execution:

- Data races, deadlocks, consecutive or missing locks and unlocks (Bug Finder)
- Unprotected shared variables (Code Prover)

Configure Analysis



If your code uses multitasking primitives from certain families, for instance, `pthread_create` for thread creation:

- In Bug Finder, the analysis detects them and extracts your multitasking model from the code.
- In Code Prover, you must enable this automatic detection explicitly. See `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.

See “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 15-7.

Alternatively, define your multitasking model through the analysis options. In the user interface of the Polyspace desktop products, the options are on the **Multitasking** node in the **Configuration** pane. Most options are common between Bug Finder and Code Prover. The multitasking analysis in Code Prover is more exhaustive about finding potentially unprotected shared variables and therefore follows a stricter model. Your code must be written in a specific format for Code Prover to successfully complete a multitasking analysis. For instance, the functions that you specify as entry points must be `void(void)` functions. However, if your code is not already written in this format, you can work around the restrictions. For details, see “Configuring Polyspace Multitasking Analysis Manually” on page 15-17.

Review Analysis Results

Bug Finder

The screenshot shows the Bug Finder interface. On the left, the 'Results List' pane displays a hierarchy of defects: Defect 249, Concurrency 9, Data race 2, and several other categories like Deadlock, Double lock, and Missing lock. The 'Data race 2' defect is selected, showing a high impact. On the right, the 'Result Details' pane provides information for the selected defect. It shows the status as 'Unreviewed' and severity as 'Unset'. A description states: 'Data race (Impact: High) Certain operations on variable 'bad_glob1' can interfere with each other and cause unpredictable values.' Below this is a table showing the conflicting operations:

Access	Access Protections	Task	File	Scope	Line
Write	No protection	bug_datarace_task10	concurrency.c	bug_datarace_task10	57
Read	No protection	bug_datarace_task20	concurrency.c	bug_datarace_task20	62

A Bug Finder analysis can find many different kinds of concurrency defects including:

- Data races, when operations on a variable from different tasks interfere with each other.
- Deadlocks or double locks, because of incorrect placement of lock and unlock functions


For the complete list, see “Concurrency Defects”. However, the analysis makes certain assumptions to avoid false positives, and might not find all data races. You can perform an initial check for data races with Bug Finder, and make a more exhaustive pass later with Code Prover.

Code Prover

The screenshot shows the Code Prover interface. On the left, the 'Results List' pane shows a hierarchy of variables: Run-time Check (Red, Gray, Orange, Green), Global Variable (Shared, Potentially unprotected, Protected, Not shared, Unused, Used non-shared). The 'Potentially unprotected variable' category is expanded, showing 'Variable: PowerLevel' for 'tasks1.'. On the right, the 'Result Details' pane shows the status as 'Unreviewed' and severity as 'Unset'. A description states: 'Potentially unprotected variable Variable 'tasks1.PowerLevel' is shared among several tasks. Some operations on variable 'tasks1.PowerLevel' have no common protection. Read by task: server1 server2 tregulate. Written by task: server1 server2 tregulate.' Below this is a table showing the conflicting operations:

Event	File	Scope	Line
Written value: -10000	main.c	main()	36
Written value: 0	tasks1.c	_init_globals()	26
Written value: [-2147483639 .. 2 ³¹ -1]	tasks2.c	Increase_PowerLevel()	19
Read value: [-2147483640 .. 2 ³¹ -1]	tasks1.c	orderregulate()	40
Read value: [-2147483640 .. 2 ³¹ -1]	tasks2.c	Increase_PowerLevel()	19
Read value: [-2147483640 .. 2 ³¹ -1]	tasks2.c	Compute_Injection()	34
Read value: [-2147483640 .. 2 ³¹ -1]	tasks2.c	Get_PowerLevel()	41

The Code Prover analysis exhaustively checks if shared global variables are protected from concurrent access. The analysis reports variables that are definitely protected in green and variables that might be unprotected in orange. See “Global Variables” (Polyspace Code Prover).

Review the results using the message on the **Result Details** pane. See a visual representation of conflicting operations using the  (graph) icon.

Differences Between Bug Finder and Code Prover

The following table summarizes the differences between the multitasking analysis in Polyspace Bug Finder and Polyspace Code Prover.

Configuration

	Bug Finder	Code Prover
Auto-detection of concurrency routines	Supported by default	Supported on option <code>Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)</code>
Constraints on main function	None	The <code>main</code> function must terminate. It cannot contain an infinite loop or a run-time error. For workarounds if there is an intentional infinite loop in <code>main</code> , see “Adapt Code for Code Prover Multitasking Analysis” (Polyspace Code Prover).
Atomic operations	Depending on the target size, certain operations are considered as atomic (non-interruptable). To consider all operations as non-atomic, use the option <code>-detect-atomic-data-race</code> . See also <code>Define Atomic Operations in Multitasking Code</code> .	All operations are considered as non-atomic.

Results

	Bug Finder	Code Prover
Concurrent unprotected access on shared variables (data races)	Shown using one of these results: <ul style="list-style-type: none"> • Data race • Data race on adjacent bit fields • Data race through standard library function call 	Shown using the result Potentially unprotected variable . Code Prover is more exhaustive when keeping track of control and data flows. Therefore, Code Prover might detect probable data races not detected with Bug Finder.
Issues with concurrency routines besides data race: <ul style="list-style-type: none"> • Deadlocks, double locks, missing unlocks, and so on. • Improper thread creation, joining or destruction. • Memory escape from threads 	Detected	Not detected

See Also**More About**

- “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 15-7
- “Configuring Polyspace Multitasking Analysis Manually” on page 15-17
- “Protections for Shared Variables in Multitasking Code” on page 15-21

Auto-Detection of Thread Creation and Critical Section in Polyspace

With Polyspace, you can analyze programs where multiple threads run concurrently. Polyspace can analyze your multitasking code for data races, deadlocks and other concurrency defects, if the analysis is aware of the concurrency model in your code. In some situations, Polyspace can detect thread creation and critical sections in your code automatically. Bug Finder detects them by default. In Code Prover, you enable automatic detection using the option `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.

For the multitasking code analysis workflow, see “Analyze Multitasking Programs in Polyspace” on page 15-2.

If your thread creation function is not detected automatically:

- You can also map the function to a thread-creation function that Polyspace can detect automatically. Use the option `-code-behavior-specifications`.
- Otherwise, you must manually model your multitasking threads by using configuration options. See “Configuring Polyspace Multitasking Analysis Manually” on page 15-17.

Multitasking Routines that Polyspace Can Detect

Polyspace can detect thread creation and critical sections if you use primitives from these groups. Polyspace recognizes calls to these routines as the creation of a new thread or as the beginning or end of a critical section.

POSIX

Thread creation: `pthread_create`

Critical section begins: `pthread_mutex_lock`

Critical section ends: `pthread_mutex_unlock`

VxWorks

Thread creation: `taskSpawn`

Critical section begins: `semTake`

Critical section ends: `semGive`

To activate automatic detection of concurrency primitives for VxWorks®, in the user interface of the Polyspace desktop products, use the VxWorks template. For more information on templates, see “Create Project in Polyspace Desktop User Interface Using Configuration Template” on page 2-13. At the command-line, use these options:

```
-D1=CPU=I80386
-D2=__GNUC__=2
-D3=__OS_VXWORKS
```

Concurrency detection is possible only if the multitasking functions are created from an entry point named `main`. If the entry point has a different name, such as `vxworks_entry_point`, do one of the following:

- Provide a main function.
- Preprocessor definitions (-D): In preprocessor definitions, set `vxworks_entry_point=main`.

Windows

Thread creation: `CreateThread`

Critical section begins: `EnterCriticalSection`

Critical section ends: `LeaveCriticalSection`

µC/OS II

Thread creation: `OSTaskCreate`

Critical section begins: `OSMutexPend`

Critical section ends: `OSMutexPost`

C++11

Thread creation: `std::thread::thread`

Critical section begins: `std::mutex::lock`

Critical section ends: `std::mutex::unlock`

For autodetection of C++11 threads, explicitly specify paths to your compiler header files or use `polyspace-configure`.

For instance, if you use `std::thread` for thread creation, explicitly specify the path to the folder containing `thread.h`.

See also “Limitations of Automatic Thread Detection” on page 15-12.

C11

Thread creation: `thrd_create`

Critical section begins: `mtx_lock`

Critical section ends: `mtx_unlock`

Example of Automatic Thread Detection

The following multitasking code models five philosophers sharing five forks. The example uses POSIX® thread creation routines and illustrates a classic example of a deadlock. Run Bug Finder on this code to see the deadlock.


```
#include "pthread.h"
#include <stdio.h>
#include <unistd.h>

pthread_mutex_t forks[5];

void* philo1(void* args)
{
    while (1) {
        printf("Philosopher 1 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[0]);
        printf("Philosopher 1 takes left fork\n");
        pthread_mutex_lock(&forks[1]);
        printf("Philosopher 1 takes right fork\n");
        printf("Philosopher 1 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[1]);
        printf("Philosopher 1 puts down right fork\n");
        pthread_mutex_unlock(&forks[0]);
        printf("Philosopher 1 puts down left fork\n");
    }
    return NULL;
}

void* philo2(void* args)
{
    while (1) {
        printf("Philosopher 2 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[1]);
        printf("Philosopher 2 takes left fork\n");
        pthread_mutex_lock(&forks[2]);
        printf("Philosopher 2 takes right fork\n");
        printf("Philosopher 2 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[2]);
        printf("Philosopher 2 puts down right fork\n");
        pthread_mutex_unlock(&forks[1]);
        printf("Philosopher 2 puts down left fork\n");
    }
    return NULL;
}

void* philo3(void* args)
{
    while (1) {
        printf("Philosopher 3 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[2]);
        printf("Philosopher 3 takes left fork\n");
        pthread_mutex_lock(&forks[3]);
        printf("Philosopher 3 takes right fork\n");
        printf("Philosopher 3 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[3]);
        printf("Philosopher 3 puts down right fork\n");
    }
}
```

```
        pthread_mutex_unlock(&forks[2]);
        printf("Philosopher 3 puts down left fork\n");
    }
    return NULL;
}

void* philo4(void* args)
{
    while (1) {
        printf("Philosopher 4 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[3]);
        printf("Philosopher 4 takes left fork\n");
        pthread_mutex_lock(&forks[4]);
        printf("Philosopher 4 takes right fork\n");
        printf("Philosopher 4 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[4]);
        printf("Philosopher 4 puts down right fork\n");
        pthread_mutex_unlock(&forks[3]);
        printf("Philosopher 4 puts down left fork\n");
    }
    return NULL;
}

void* philo5(void* args)
{
    while (1) {
        printf("Philosopher 5 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[4]);
        printf("Philosopher 5 takes left fork\n");
        pthread_mutex_lock(&forks[0]);
        printf("Philosopher 5 takes right fork\n");
        printf("Philosopher 5 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[0]);
        printf("Philosopher 5 puts down right fork\n");
        pthread_mutex_unlock(&forks[4]);
        printf("Philosopher 5 puts down left fork\n");
    }
    return NULL;
}

int main(void)
{
    pthread_t ph[5];
    pthread_create(&ph[0], NULL, philo1, NULL);
    pthread_create(&ph[1], NULL, philo2, NULL);
    pthread_create(&ph[2], NULL, philo3, NULL);
    pthread_create(&ph[3], NULL, philo4, NULL);
    pthread_create(&ph[4], NULL, philo5, NULL);

    pthread_join(ph[0], NULL);
    pthread_join(ph[1], NULL);
    pthread_join(ph[2], NULL);
    pthread_join(ph[3], NULL);
    pthread_join(ph[4], NULL);
}
```

```

return 1;
}

```

Each philosopher needs two forks to eat, a right and a left fork. The functions `philo1`, `philo2`, `philo3`, `philo4`, and `philo5` represent the philosophers. Each function requires two `pthread_mutex_t` resources, representing the two forks required to eat. All five functions run at the same time in five concurrent threads.

However, a deadlock occurs in this example. When each philosopher picks up their first fork (each thread locks one `pthread_mutex_t` resource), all the forks are being used. So, the philosophers (threads) wait for their second fork (second `pthread_mutex_t` resource) to become available. However, all the forks (resources) are being held by the waiting philosophers (threads), causing a deadlock.

Naming Convention for Automatically Detected Threads

If you use a function such as `pthread_create()` to create new threads (tasks), each thread is associated with a unique identifier. For instance, in this example, two threads are created with identifiers `id1` and `id2`.

```

pthread_t* id1, id2;

void main()
{
    pthread_create(id1, NULL, start_routine, NULL);
    pthread_create(id2, NULL, start_routine, NULL);
}

```

If a data race occurs between the threads, the analysis can detect it. When displaying the results, the threads are indicated as `task_id`, where `id` is the identifier associated with the thread. In the preceding example, the threads are identified as `task_id1` and `task_id2`.

If a thread identifier is:

- Local to a function, the thread name shows the function.

For instance, the thread created below appears as `task_f:id`

```

void f(void)
{
    pthread_t* id;
    pthread_create(id, NULL, start_routine, NULL);
}

```

- A field of a structure, the thread name shows the structure.

For instance, the thread created below appears as `task_a#id`

```

struct {pthread_t* id; int x;} a;
pthread_create(a.id, NULL, start_routine, NULL);

```

- An array member, the thread name shows the array.

For instance, the thread created below appears as `task_tab[1]`.

```
pthread_t* tab[10];
pthread_create(tab[1],NULL,start_routine,NULL);
```

If you create two threads with distinct thread identifiers, but you use the same local variable name for the thread identifiers, the name of the second thread is modified to distinguish it from the first thread. For instance, the threads below appear as `task_func:id` and `task_func:id:1`.

```
void func()
{
    {
        pthread_t id;
        pthread_create(&id, NULL, &task, NULL);
    }
    {
        pthread_t id;
        pthread_create(&id, NULL, &task, NULL);
    }
}
```

Limitations of Automatic Thread Detection

The multitasking model extracted by Polyspace does not include some features. Polyspace cannot model:

- Thread priorities and attributes — Ignored by Polyspace.
- Recursive semaphores.
- Unbounded thread identifiers, such as `extern pthread_t ids[]` — Warning.
- Calls to concurrency primitive through high-order calls — Warning.
- Aliases on thread identifiers — Polyspace over-approximates when the alias is used.
- Termination of threads — Polyspace ignores `pthread_join` and `thrd_join`. Polyspace replaces `pthread_exit` and `thrd_exit` by a standard `exit`.
- (Polyspace Bug Finder only) Creation of multiple threads through multiple calls to the same function with different pointer arguments.

Example

In this example, Polyspace considers that only one thread is created.

```
pthread_t id1, id2;
void start(pthread_t* id)
{
    pthread_create(id, NULL, start_routine, NULL);
}
void main()
{
    start(&id1);
    start(&id2);
}
```

- (Polyspace Code Prover only) Shared local variables — Only global variables are considered shared. If a local variable is accessed by multiple threads, the analysis does not take into account the shared nature of the variable.

Example

In this example, the analysis does not take into account that the local variable `x` can be accessed by both `task1` and `task2` (after the new thread is created).

```
#include <pthread.h>
#include <stdlib.h>

void* task2(void* args)
{
    int* x = (int*) args;
    *x = 1;
    return (void*)x;
}

void task1()
{
    int x;
    x = 2;
    pthread_t id;
    (void)pthread_create(&id, NULL, task2, (void*) &x);
    /* x (local var) passed to task2 */
    x = 3 ;

    /* Unknown thread priority means x = 1 OR x = 3.*/
    /* However, the analysis considers x = 3 */
    /* Assertion below is green */
    assert(x == 3);
}

int main(void)
{
    task1();
    return 0;
}
```

- (Polyspace Code Prover only) Shared dynamic memory — Only global variables are considered shared. If a dynamically allocated memory region is accessed by multiple threads, the analysis does not take into account its shared nature.

Example

In this example, the analysis does not take into account that `lx` points to a shared memory region. The region can be accessed by both `task1` and `task2` (after the new thread is created). The Code Prover analysis also reports `lx` as a non-shared variable.

```
#include <pthread.h>
#include <stdlib.h>

static int* lx;

void* task2(void* args)
{
    int* x = (int*) args;
    *x = 1;
    return (void*)x;
}

void task1()
{
    pthread_t id;
    lx = (int*)malloc(sizeof(int));

    if (lx == NULL) exit(1);

    (void)pthread_create(&id, NULL, task2, (void*) lx);

    *lx = 3 ;

    /* Unknown thread priority means *lx = 1 OR *lx = 3.*/
    /* However, the analysis considers *lx = 3 */
    /* Assertion below is green */
    assert(*lx == 3);
}

int main(void)
{
    task1();
    return 0;
}
```

- Number of tasks created with `CreateThread` when `threadId` is set to `NULL`— When you create multiple threads that execute the same function, if the last argument of `CreateThread` is `NULL`, Polyspace only detects one instance of this function, or task.

Example

In this example, Polyspace detects only one instance of `thread_function1()`, but 10 instances of `thread_function2()`.

```

#include <windows.h>

#define MAX_LOOP_THREADS 10

DWORD WINAPI thread_function1(LPVOID data) {}
DWORD WINAPI thread_function2(LPVOID data) {}

HANDLE hds1[MAX_LOOP_THREADS];
HANDLE hds2[MAX_LOOP_THREADS];
DWORD threadId[MAX_LOOP_THREADS];

int main(void)
{
    for (int i = 0; i < MAX_LOOP_THREADS; i++) {
        hds1[i] = CreateThread(NULL, 0, thread_function1, NULL, 0, NULL);
        hds2[i] = CreateThread(NULL, 0, thread_function2, NULL, 0, &threadId[i]);
    }

    return 0;
}

```

- (C++11 only) If you use lambda expressions as start functions during thread creation, Polyspace does not detect shared variables in the lambda expressions.

Example

In this example, Polyspace does not detect that the variable `y` used in the lambda expressions is shared between two threads. As a result, Bug Finder, for instance, does not show a **Data race** defect.

```

#include <thread>
int y;
int main() {
    std::thread t1([] {y++;});
    std::thread t2([] {y++;});
    t1.join();
    t2.join();
    return 0;
}

```

- (C++11 threads with Polyspace Code Prover only) String literals as thread function argument — Code Prover shows a red **Illegally dereferenced pointer** error if the thread function has an `std::string&` parameter and you pass a string literal argument.

Example

In this example, the thread function `foo` has an `std::string&` parameter. When starting a thread, a string literal is passed as argument to this function, which undergoes an implicit conversion to `std::string` type. Code Prover loses track of the original string literal in this conversion. Therefore, a dashed red underline appears on `operator<<` in the body of `foo` and a red **Illegally dereferenced pointer** check in the body of `operator<<`.

```
#include <iostream>
#include <thread>

using namespace std;

void foo(const std::string& f) {
    std::cout << f;
}

void main() {
    std::thread t1(foo,"foo_arg");
}
```

To work around this issue, assign the string literal to a temporary variable and pass the variable as argument to the thread function.

```
#include <iostream>
#include <thread>

using namespace std;

void foo(const std::string& f) {
    std::cout << f;
}

void main() {
    std::string str = "foo_arg";
    std::thread t1(foo, str);
}
```

See Also

Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection) | -code-behavior-specifications

More About

- “Analyze Multitasking Programs in Polyspace” on page 15-2
- “Configuring Polyspace Multitasking Analysis Manually” on page 15-17

Configuring Polyspace Multitasking Analysis Manually

With Polyspace, you can analyze programs where multiple threads run concurrently. In some situations, Polyspace can detect thread creation and critical sections in your code automatically. See “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 15-7.

If your code has functions that are intended for concurrent execution, but that cannot be detected automatically, you must specify them before analysis. If these functions operate on a common variable, you must also specify protection mechanisms for those operations.

For the multitasking code analysis workflow, see “Analyze Multitasking Programs in Polyspace” on page 15-2.

Specify Options for Multitasking Analysis

Use these options to specify cyclic tasks, interrupts and protections for shared variables. In the user interface of the Polyspace desktop products, the options are on the **Multitasking** node in the **Configuration** pane. The following options can be used in both a Bug Finder and Code Prover analysis:

- **Tasks (-entry-points)**: Specify noncyclic entry point functions.
Do not specify `main`. Polyspace implicitly considers `main` as an entry point function.
- **Critical section details (-critical-section-begin -critical-section-end)**: Specify functions that begin and end critical sections.
- **Temporally exclusive tasks (-temporal-exclusions-file)**: Specify groups of functions that are temporally exclusive.

A Polyspace analysis supports four levels of task priorities. That is, the analysis can take into consideration the fact that certain tasks cannot be interrupted by tasks with lower priorities. You can use these options to indicate task priorities:

- **Cyclic tasks (-cyclic-tasks)**: Specify functions that are scheduled at periodic intervals.
- **Interrupts (-interrupts)**: Specify functions that can run asynchronously.
- **-preemptable-interrupts**: Specify functions that have lower priority than interrupts, but higher priority than tasks (preemptable or non-preemptable).
- **-non-preemptable-tasks**: Specify functions that have higher priority than tasks, but lower priority than interrupts (preemptable or non-preemptable).
- **Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)**: Specify functions that disable and reenable interrupts.

For an example of using priorities, see “Protections for Shared Variables in Multitasking Code” on page 15-21.

Adapt Code for Code Prover Multitasking Analysis

The multitasking analysis in Code Prover is more exhaustive about finding potentially unprotected shared variables and therefore follows a strict model.

Tasks and interrupts must be void(void) functions.

Functions that you specify as tasks and interrupts must have the prototype:

```
void func(void);
```

Suppose you want to specify a function `func` that takes `int` arguments and has return type `int`:

```
int func(int);
```

Define a wrapper void-void function that calls `func` with a volatile value. Specify this wrapper function as a task or interrupt.

```
void func_wrapper() {  
    volatile int arg;  
    (void)func(arg);  
}
```

You can save the wrapper function definition along with a declaration of the original function in a separate file and add this file to the analysis.

The main function must end.

Code Prover assumes that the `main` function ends before all tasks and interrupts begin. If the `main` function contains an infinite loop or run-time error, the tasks and interrupts are not analyzed. If you see that there are no checks in your tasks and interrupts, look for a token underlined in dashed red to identify the issue in the `main` function. See “Reasons for Unchecked Code” (Polyspace Code Prover).

Suppose you want to specify the `main` function as a cyclic task.

```
void performTask1Cycle(void);  
void performTask2Cycle(void);
```

```
void main() {  
    while(1) {  
        performTask1Cycle();  
    }  
}
```

```
void task2() {  
    while(1) {  
        performTask2Cycle();  
    }  
}
```

Replace the definition of `main` with:

```
#ifdef POLYSPACE  
void main() {  
}  
void task1() {  
    while(1) {  
        performTask1Cycle();  
    }  
}  
#else  
void main() {  
    while(1) {  
        performTask1Cycle();  
    }  
}
```

```

    }
}
#endif

```

The replacement defines an empty `main` and places the content of `main` into another function `task1` if a macro `POLYSPACE` is defined. Define the macro `POLYSPACE` using the option `Preprocessor definitions (-D)` and specify `task1` for the option `Tasks (-entry-points)`.

This assumption does not apply to automatically detected threads. For instance, a `main` function can create threads using `pthread_create`.

The Polyspace multitasking analysis assumes that a task or interrupt cannot interrupt itself.

All tasks and interrupts can run any number of times in any sequence.

The Code Prover analysis considers that all tasks and interrupts can run any number of times in any sequence.

Suppose in this example, you specify `reset` and `inc` as cyclic tasks. The analysis shows an overflow on the operation `var+=2`.

```

void reset(void) {
    var=0;
}

void inc(void) {
    var+=2;
}

```

Suppose you want to model a scheduling of tasks such that `reset` executes after `inc` has executed five times. Write a wrapper function that implements this sequence. Specify this new function as a cyclic task instead of `reset` and `inc`.

```

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        inc();
        inc();
        inc();
        inc();
        inc();
        reset();
    }
}

```

Suppose you want to model a scheduling of tasks such that `reset` executes after `inc` has executed zero to five times. Write a wrapper function that implements this sequence. Specify this new function as a cyclic task instead of `reset` and `inc`.

```

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        if(randomValue)
            inc();
        if(randomValue)
            inc();
        if(randomValue)

```

```
    inc();  
    if(randomValue)  
        inc();  
    if(randomValue)  
        inc();  
    reset();  
    }  
}
```

See Also

More About

- “Analyze Multitasking Programs in Polyspace” on page 15-2
- “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 15-7

Protections for Shared Variables in Multitasking Code

If your code is intended for multitasking, tasks in your code can access a common shared variable. To prevent data races, you can protect read and write operations on the variable. This topic shows the various protection mechanisms that Polyspace can recognize.

Detect Unprotected Access

Access	Access Protections	Task	File	Scope	Line
Write	No protection	bug_datarace_task10	concurrency.c	bug_datarace_task10	57
Read	No protection	bug_datarace_task20	concurrency.c	bug_datarace_task20	62

You can detect an unprotected access using either Bug Finder or Code Prover. Code Prover is more exhaustive and proves if a shared variable is protected from concurrent access.

- Bug Finder detects an unprotected access using the result **Data race**. See [Data race](#).
- Code Prover detects an unprotected access using the result **Shared unprotected global variable**. See [Potentially unprotected variable](#).

Suppose you analyze this code, specifying `signal_handler_1` and `signal_handler_2` as cyclic tasks. Use the analysis option `Cyclic tasks (-cyclic-tasks)`.

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void signal_handler_1(void) {
    reset();
    inc();
    inc();
}

void signal_handler_2(void) {
    shared_var = INT_MAX;
}

void main() {
}
```

Bug Finder shows a data race on `shared_var`. Code Prover shows that `shared_var` is a potentially unprotected shared variable. Code Prover also shows that the operation `shared_var += 2` can overflow. The overflow occurs if the call to `inc` in `signal_handler_1` immediately follows the operation `shared_var = INT_MAX` in `signal_handler_2`.

Protect Using Critical Sections

One possible solution is to protect operations on shared variables using critical sections.

In the preceding example, modify your code so that operations on `shared_var` are in the same critical section. Use the functions `take_semaphore` and `give_semaphore` to begin and end the critical sections. To specify these functions that begin and end critical sections, use the analysis options `Critical section details (-critical-section-begin -critical-section-end)`.

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

/* Declare lock and unlock functions */
void take_semaphore(void);
void give_semaphore(void);

void signal_handler_1() {
    /* Begin critical section */
    take_semaphore();
    reset();
    inc();
    inc();
    /* End critical section */
    give_semaphore();
}

void signal_handler_2() {
    /* Begin critical section */
    take_semaphore();
    shared_var = INT_MAX;
    /* End critical section */
    give_semaphore();
}

void main() {
}
```

You do not see the data race in Bug Finder. Code Prover proves that the shared variable is protected. You also do not see the overflow because the call to `reset()` in `signal_handler_1` always precedes calls to `inc()`.

You can also use primitives such as the POSIX functions `pthread_mutex_lock` and `pthread_mutex_unlock` to begin and end critical sections. For a list of primitives that Polyspace can detect automatically, see “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 15-7.

Protect Using Temporally Exclusive Tasks

Another possible solution is to specify a group of tasks as temporally exclusive. Temporally exclusive tasks cannot interrupt each other.

In the preceding example, specify that `signal_handler_1` and `signal_handler_2` are temporally exclusive. Use the option `Temporally exclusive tasks (-temporal-exclusions-file)`.

You do not see the data race in Bug Finder. Code Prover proves that the shared variable is protected. You also do not see the overflow because the call to `reset()` in `signal_handler_1` always precedes calls to `inc()`.

Protect Using Priorities

Another possible solution is to specify that one task has higher priority over another.

In the preceding example, specify that `signal_handler_1` is an interrupt. Retain `signal_handler_2` as a cyclic task. Use the options `Cyclic tasks (-cyclic-tasks)` and `Interrupts (-interrupts)`.

Bug Finder does not show the data race defect anymore. The reason is this:

- The operation `shared_var = INT_MAX` in `signal_handler_2` is atomic. Therefore, the operations in `signal_handler_1` cannot interrupt it.
- The operations in `signal_handler_1` cannot be interrupted by the operation in `signal_handler_2` because `signal_handler_1` has higher priority.

You can specify up to four different priorities with these options (with highest priority listed first):

- `Interrupts (-interrupts)`
- `-preemptable-interrupts`
- `-non-preemptable-tasks`
- `Cyclic tasks (-cyclic-tasks)`

A task with higher priority is atomic with respect to a task with lower priority. Note that if you use the option `-detect-atomic-data-race`, the analysis ignores the difference in priorities and continues to show the data race. See also “Define Task Priorities for Data Race Detection in Polyspace” on page 15-28.

Code Prover does not consider atomicity of operations, so it continues to show `shared_var` as a potentially unprotected variable (the operations in `signal_handler_1` can still interrupt the operations in `signal_handler_2`). Code Prover shows `shared_var` as protected only when you specify both `signal_handler_1` and `signal_handler_2` as interrupts.

Protect By Disabling Interrupts

In a Bug Finder analysis, you can protect a group of operations by disabling all tasks and interrupts other than the current one.

Use the option `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)` to specify a routine that disables all interruption when called, and a routine that reenables them. The disabling routine disables preemption by all:

- Non-cyclic tasks.
See `Tasks (-entry-points)`.
- Cyclic tasks.
See `Cyclic tasks (-cyclic-tasks)`.
- Interrupts.
See `Interrupts (-interrupts)`.

In other words, the analysis considers that the body of operations between the disabling routine and the enabling routine is atomic and not interruptible at all.

After you call a routine to disable interrupts, all subsequent operations are atomic until you call the other routine to reenable interrupts. The operations are atomic with respect to operations in all other tasks.

See Also

More About

- “Analyze Multitasking Programs in Polyspace” on page 15-2
- “Define Atomic Operations in Multitasking Code” on page 15-25

Define Atomic Operations in Multitasking Code

In code with multiple threads, you can use Polyspace Bug Finder to detect data races or Polyspace Code Prover to list potentially unprotected shared variables.

To determine if a variable shared between multiple threads is protected against concurrent access, Polyspace checks if the operations on the variable are atomic.

Nonatomic Operations

If an operation is nonatomic, Polyspace considers that the operation involves multiple steps. These steps do not need to occur together and can be interrupted by operations in other threads.

For instance, consider these two operations in two different threads:

- Thread 1: `var++`;

This operation is nonatomic because it takes place in three steps: reading `var`, incrementing `var`, and writing back `var`.

- Thread 2: `var = 0`;

This operation is atomic if the size of `var` is less than the word size on the target. See details below for how Polyspace determines the word size.

If the two operations are not protected (by using, for instance, critical sections), the operation in the second thread can interrupt the operation in the first thread. If the interruption happens after `var` is incremented in the first thread but before the incremented value is written back, you can see unexpected results.

What Polyspace Considers as Nonatomic

Code Prover considers all operations as nonatomic unless you protect them, for instance, by using critical sections. See “Define Specific Operations as Atomic” on page 15-26.

Bug Finder considers an operation as nonatomic if it can translate into more than one machine instruction. For instance:

- The operation can involve both a read and write operation. For example, `var++` involves reading the value of `var`, increasing the value by one and writing the increased value back to `var`.
- The operation can involve a 64-bit variable on a 32-bit target. For example, the operation

```
long long var1, var2;
var1=var2;
```

involves two steps in copying the content of `var2` to `var1` on certain targets.

Polyspace uses the **Pointer** size for your **Target processor type** as the threshold to compute atomicity. For instance, if you use `i386` as your **Target processor type**, the **Pointer** size is 32 bits and **Long long** and **Double** sizes are both 64 bits. Therefore, Polyspace considers copying one `long long` or `double` variable to another as nonatomic.

See also `Target processor type (-target)`.

- The operation can involve writing the return value of a function call to a shared variable. For example, the operation `x=func()` involves calling `func` and writing the return value of `func` to `x`.

To detect data races where at least one of the two interrupting operations is nonatomic, enable the Bug Finder checker `Data race`. To remove this constraint on the checker, use the option `-detect-atomic-data-race`.

Define Specific Operations as Atomic

You might want to define a group of operations as atomic. This group of operations cannot be interrupted by operations in another thread or task.

Use one of these techniques:

- **Critical sections**

Protect a group of operations with critical sections.

A critical section begins and ends with calls to specific functions. You can use a predefined set of primitives to begin or end critical sections, or use your own functions.

A group of operations in a critical section are atomic with respect to another group of operations that are in the same critical section (that is, having the same beginning and ending function).

Specify critical sections using the option `Critical section details` (`-critical-section-begin` `-critical-section-end`).

- **Temporally exclusive tasks**

Protect a group of operations by specifying certain tasks as temporally exclusive.

If a group of tasks are temporally exclusive, all operations in one task are atomic with respect to operations in the other tasks.

Specify temporal exclusion using the option `Temporally exclusive tasks` (`-temporal-exclusions-file`).

- **Task priorities**

Protect a group of operations by specifying that certain tasks have higher priorities. For instance, interrupts have higher priorities over cyclic tasks.

You can specify up to four different priorities with these options (with highest priority listed first):

- `Interrupts` (`-interrupts`)
- `-preemptable-interrupts`
- `-non-preemptable-tasks`
- `Cyclic tasks` (`-cyclic-tasks`)

All operations in a task with higher priority are atomic with respect to operations in tasks with lower priorities. See also “Define Task Priorities for Data Race Detection in Polyspace” on page 15-28.

- **Routine disabling interrupts** (Bug Finder only)

Protect a group of operations by disabling all interrupts. Use the option `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

After you call a routine to disable interrupts, all subsequent operations are atomic until you call another routine to reenable interrupts. The operations are atomic with respect to operations in all other tasks.

For a tutorial, see “Protections for Shared Variables in Multitasking Code” on page 15-21.

See Also

`Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

More About

- “Analyze Multitasking Programs in Polyspace” on page 15-2
- “Protections for Shared Variables in Multitasking Code” on page 15-21

Define Task Priorities for Data Race Detection in Polyspace

Bug Finder detects data races between concurrent tasks. One of the ways you can fix data races is by specifying that certain tasks have higher priorities over others. A task with higher priority is atomic with respect to tasks with lower priority and cannot be interrupted by those tasks.

Emulating Task Priorities

You can specify up to four different priorities with these options (with highest priority listed first):

- Interrupts (nonpreemptable): Use option `Interrupts (-interrupts)`.
- Interrupts (preemptable): Use options `Interrupts (-interrupts)` and `-preemptable-interrupts`.
- Cyclic tasks (nonpreemptable): Use options `Cyclic tasks (-cyclic-tasks)` and `-non-preemptable-tasks`.

You can also define preemptable noncyclic tasks with the option `Tasks (-entry-points)` and `-non-preemptable-tasks`.

- Cyclic tasks (preemptable): Use option `Cyclic tasks (-cyclic-tasks)`.

You can also define noncyclic tasks with the option `Tasks (-entry-points)`.

For instance, interrupts have the highest priority and cannot be preempted by other tasks. To define a class of interrupts that can be preempted, lower their priority by making them preemptable.

Examples of Task Priorities

Consider this example with three tasks. A variable `var` is shared between the two tasks `task1` and `task2` without any protection such as a critical section. Depending on the priorities of `task1` and `task2`, Bug Finder shows a data race. The third task is not relevant for the example (and is added only to include a critical section, otherwise data race detection is disabled).

```
int var;

void begin_critical_section(void);
void end_critical_section(void);

void task1(void) {
    var++;
}

void task2(void) {
    var = 0;
}

void task3(void){
    begin_critical_section();
    /* Some atomic operation */
    end_critical_section();
}
```

Adjust the priorities of `task1` and `task2` and see whether a data race is detected. For instance:

1 Configure these multitasking options:

- `Interrupts (-interrupts)`: Specify `task1` and `task2` as interrupts.
- `Cyclic tasks (-cyclic-tasks)`: Specify `task3` as a cyclic task.
- `Critical section details (-critical-section-begin -critical-section-end)`: Specify `begin_critical_section` as a function beginning a critical section and `end_critical_section` as a function ending a critical section.

2 Run Bug Finder.

You do not see a data race. Since `task1` and `task2` are nonpreemptable interrupts, the shared variable cannot be accessed concurrently.

3 Change `task1` to a preemptable interrupt by using the option `-preemptable-interrupts`.

4 Run Bug Finder again. You now see a data race on the shared variable `var`.

Further Explorations

Modify this example in the following ways and see the effect of the modification:

- Change the priorities of `task1` and `task2`.

For instance, you can leave `task1` as a nonpreemptable interrupt but change `task2` to a preemptable interrupt by using the option `-preemptable-interrupts`.

The data race disappears. The reason is:

- `task1` has higher priority and cannot be interrupted by `task2`.
- The operation in `task2` is atomic and cannot be interrupted by `task1`.
- Specify the option `-detect-atomic-data-race`.

You see the data race again. The checker considers all operations as potentially nonatomic and the operation in `task2` can now be interrupted by the higher priority operation in `task1`.

Try other modifications to the analysis options and see the result of the checkers.

Effect of Task Priorities in Code Prover

The options to specify task priorities are also accepted in Code Prover. However, Code Prover considers all operations as potentially non-atomic and interruptible. This overapproximation can lead to situations where the task priority specifications appear to be ignored.

For instance, in the preceding example, if you run Code Prover, the overapproximation can lead to false positives.

- If you specify both `task1` and `task2` as nonpreemptable interrupts, the shared variable `var` appears as a green **Shared protected global variable**. This is a sound result since both tasks cannot be interrupted.
- If you specify that `task1` has lower priority than `task2`, the shared variable `var` appears as an orange **Potentially unprotected variable**. This is a sound and precise result since the operation `var++` in `task1` is nonatomic and involves more than one machine instruction. The operation can be interrupted by the operation `var = 0` in `task2`.

- If you specify that `task1` has higher priority than `task2`, the shared variable `var` still appears as an orange **Potentially unprotected variable**. This is a sound but imprecise result:
 - The operation `var++` in `task1` cannot be interrupted because of the higher priority of `task1`.
 - The operation `var = 0` in `task2` cannot be interrupted because it is atomic.

However, because Code Prover considers all operations as potentially non-atomic, it considers `var = 0` in `task2` as interruptible and therefore continues to show `var` as potentially unprotected.

See Also

Polyspace Analysis Options

Interrupts (-interrupts) | -preemptable-interrupts | -non-preemptable-tasks |
Cyclic tasks (-cyclic-tasks)

Polyspace Results

Data race | Potentially unprotected variable | Shared variable

More About

- “Analyze Multitasking Programs in Polyspace” on page 15-2
- “Protections for Shared Variables in Multitasking Code” on page 15-21
- “Define Atomic Operations in Multitasking Code” on page 15-25

Define Critical Sections with Functions That Take Arguments

When verifying multitasking code, Polyspace considers that a critical section lies between calls to a lock and unlock function.

```
lock();
/* Critical section code */
unlock();
```

A group of operations in a critical section are atomic with respect to another group of operations that are in the same critical section (that is, having the same lock and unlock function). See also “Define Atomic Operations in Multitasking Code” on page 15-25.

Polyspace Assumption on Functions Defining Critical Sections

Polyspace ignores arguments to functions that begin and end critical sections.

For instance, Polyspace treats the two code sections below as the same critical section if you specify `my_task_1` and `my_task_2` as entry points, `my_lock` as the lock function and `my_unlock` as the unlock function.

```
int shared_var;

void my_lock(int);
void my_unlock(int);

void my_task_1() {
    my_lock(1);
    /* Critical section code */
    shared_var=0;
    my_unlock(1);
}

void my_task_2() {
    my_lock(2);
    /* Critical section code */
    shared_var++;
    my_unlock(2);
}
```

As a result, the analysis considers that these two sections are protected from interrupting each other even though they might not be protected. For instance, Bug Finder does not detect the data race on `shared_var`.

Often, the function arguments can be determined only at run time. Since Polyspace models the critical sections prior to the static analysis and run-time error checking phase, the analysis cannot determine if the function arguments are different and ignores the arguments.

Adapt Polyspace Analysis to Lock and Unlock Functions with Arguments

When the arguments to the functions defining critical sections are compile-time constants, you can adapt the analysis to work around the Polyspace assumption.

For instance, you can use Polyspace analysis options so that the code in the preceding example appears to Polyspace as shown here.

```
int shared_var;

void my_lock_1(void);
void my_lock_2(void);
void my_unlock_1(void);
void my_unlock_2(void);

void my_task_1() {
    my_lock_1();
    /* Critical section code */
    shared_var=0;
    my_unlock_1();
}

void my_task_2() {
    my_lock_2();
    /* Critical section code */
    shared_var++;
    my_unlock_2();
}
```

If you then specify `my_lock_1` and `my_lock_2` as the lock functions and `my_unlock_1` and `my_unlock_2` as the unlock functions, the analysis recognizes the two sections of code as part of different critical sections. For instance, Bug Finder detects a data race on `shared_var`.

To adapt the analysis for lock and unlock functions that take compile-time constants as arguments:

- 1 In a header file `common_polyspace_include.h`, convert the function arguments into extensions of the function name with `#define`-s. In addition, provide a declaration for the new functions.

For instance, for the preceding example, use these `#define`-s and declarations:

```
#define my_lock(X) my_lock_##X()
#define my_unlock(X) my_unlock_##X()

void my_lock_1(void);
void my_lock_2(void);
void my_unlock_1(void);
void my_unlock_2(void);
```

- 2 Specify the file name `common_polyspace_include.h` as argument for the option `Include (-include)`.

The analysis considers this header file as `#include`-d in all source files that are analyzed.

- 3 Specify the new function names as functions beginning and ending critical sections. Use the options `Critical section details (-critical-section-begin -critical-section-end)`.

See Also

`Critical section details (-critical-section-begin -critical-section-end)`

More About

- “Protections for Shared Variables in Multitasking Code” on page 15-21

Configure Coding Rules Checking and Code Metrics Computation

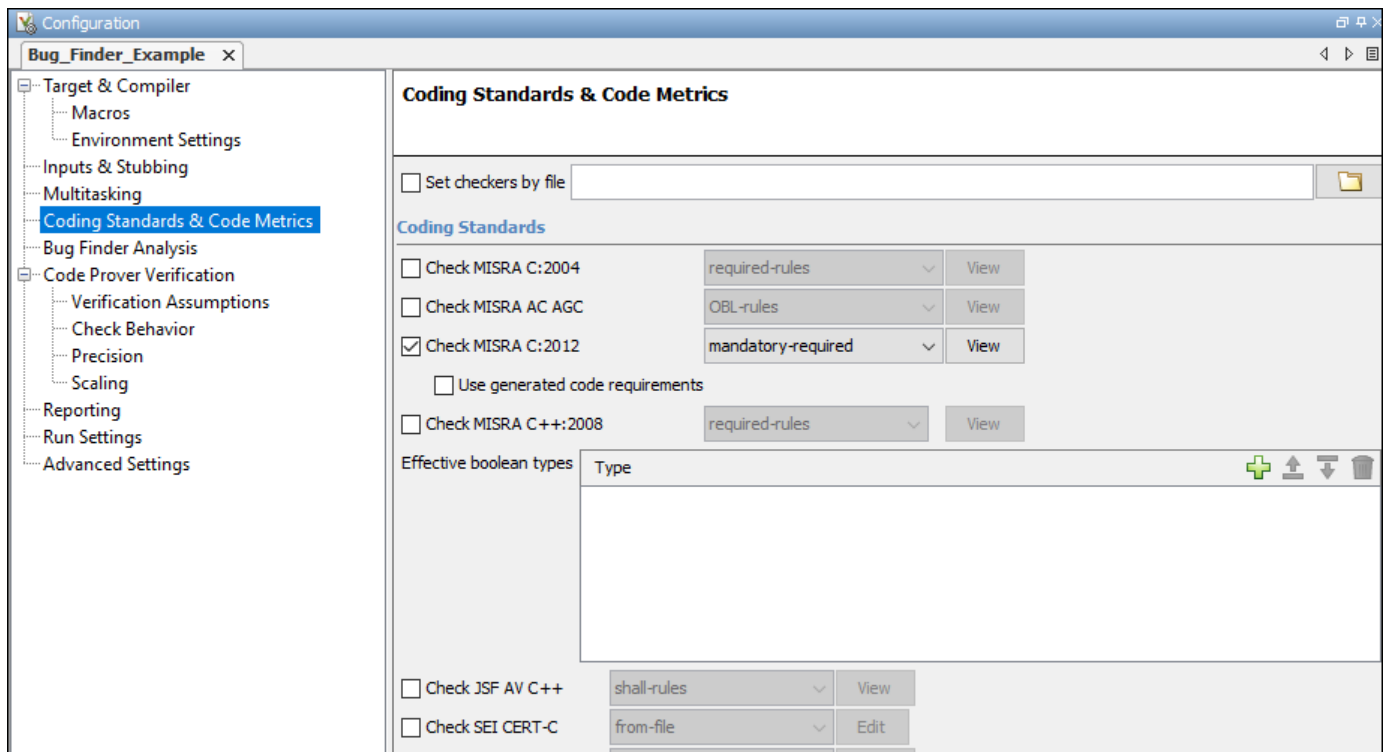
Check for and Review Coding Standard Violations

With Polyspace, you can check your C/C++ code for violations of coding rules such as MISRA C:2012 rules. Adhering to coding rules can reduce the number of defects and improve the quality of your code.

Polyspace can detect coding rule violations for these standards:

- MISRA C:2004
- MISRA C:2012
- MISRA C++
- JSF AV C++
- AUTOSAR C++14 (*Bug Finder only*)
- CERT C (*Bug Finder only*)
- CERT C++ (*Bug Finder only*)
- CWE™ (*Bug Finder only*)
- ISO®/IEC TS 17961 (*Bug Finder only*)
- Guidelines (*Bug Finder only*)

Configure Coding Rules Checking



Specify Standard and Predefined Checker Subsets

Specify the coding rules through Polyspace analysis options. When you run Bug Finder, the analysis looks for coding rule violations in addition to other checks. You can disable the other checks and look for coding rule violations only.

In the Polyspace user interface (desktop products), the options are on the **Configuration** pane under the **Coding Standards & Code Metrics** node.

For C code, use one of these options:

- Check MISRA C:2004 (-misra2)

For generated code, enable the option specific to generated code.

- Check MISRA C:2012 (-misra3)

For generated code, enable the option specific to generated code.

- Check SEI CERT-C (-cert-c)
- Check ISO/IEC TS 17961 (-iso-17961)
- Check guidelines (-guidelines)
- Check CWE (-cwe)

For C++ code, use one of these options:

- Check MISRA C++:2008 (-misra-cpp)
- Check JSF AV C++ rules (-jsf-coding-rules)
- Check AUTOSAR C++ 14 (-autosar-cpp14)
- Check SEI CERT-C++ (-cert-cpp)
- Check guidelines (-guidelines)
- Check CWE (-cwe)

You can specify a predefined subset of rules, for instance, mandatory for MISRA C:2012. These subsets are typically defined by the standard.

You can also define naming conventions for identifiers using regular expressions. See “Create Custom Coding Rules” on page 16-48.

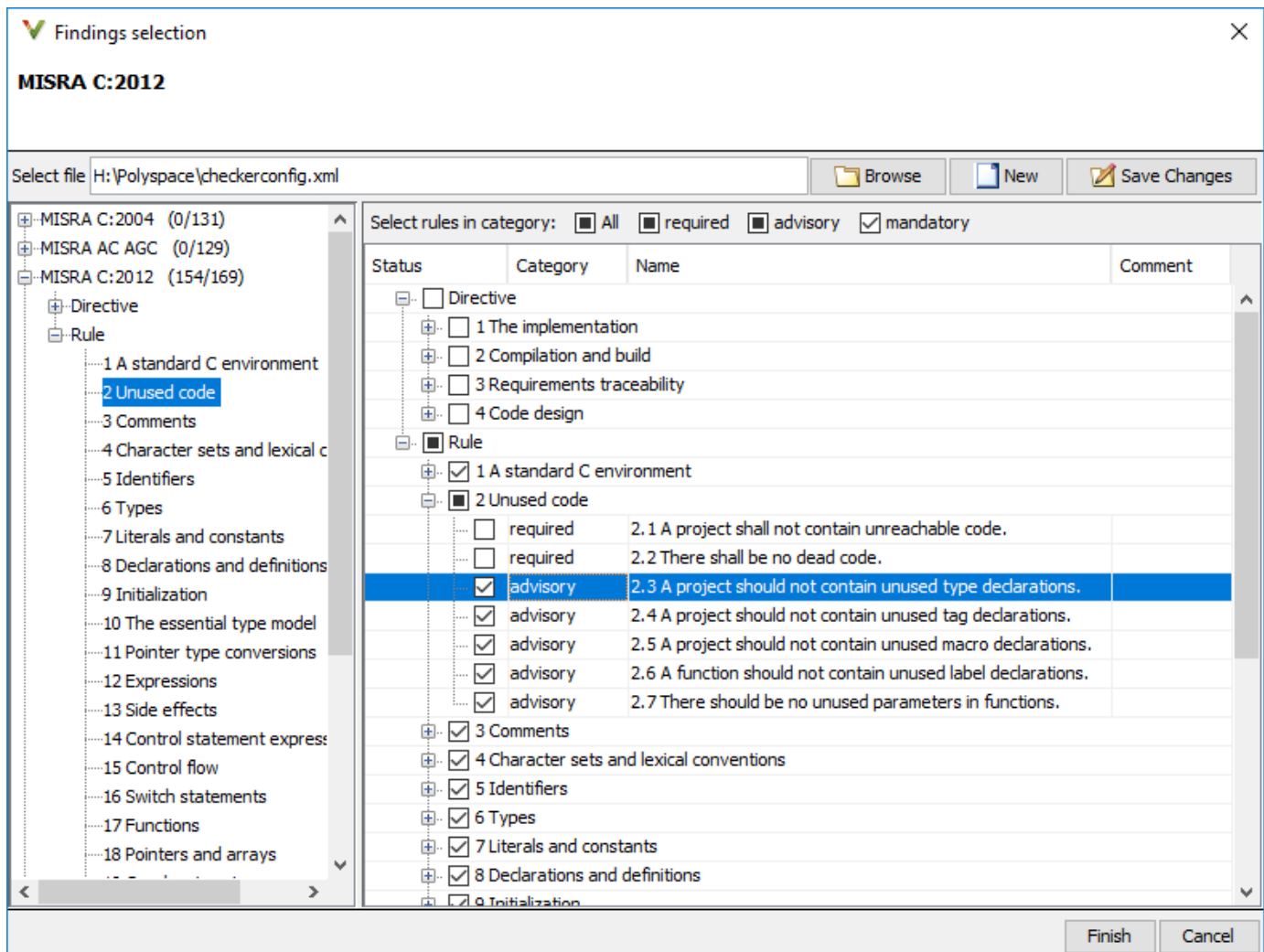
Customize Checker Subsets

Instead of the predefined subsets, you can specify your own subset of rules from a coding standard.

User Interface (Desktop Products Only)

- 1 Select the coding standard. From the drop-down list for the subset of rules, select from-file. Click **Edit**.
- 2 In the **Checkers selection** window, the coding standard is highlighted on the left pane. On the right pane, select the rules that you want to include in your analysis.
 - When selecting **Guidelines > Software Complexity** checkers, review their thresholds. If the default thresholds are not acceptable, specify a suitable threshold in the **Threshold** column. See Check guidelines (-guidelines).

- When selecting **Custom** rules, review the **Pattern** and **Convention** for the rules. See Check custom rules (-custom-rules).



When you save the rule selections, the configuration is saved in an XML file that you can reuse for multiple analyses. The same file contains rules selected for all coding standards. You can reuse this file across multiple projects to enforce common coding standards in a team or organization. To reuse this file in another project in the Polyspace user interface:

- Choose a coding standard in the project configuration. From the drop-down list for the subset of rules, select from-file.
- Click **Edit** and browse to the file location. Alternatively, enter the file name as argument for the option Set checkers by file (-checkers-selection-file).

Command Line

With the Polyspace desktop products, you can create a coding standard XML file in the user interface and then use this file for command-line analysis. Provide this XML file with the option Set checkers by file (-checkers-selection-file).

With the Polyspace Server products, you have to create a coding standard XML from scratch. Depending on the standard that you want to enable, make a writeable copy of one of the files in *polyspaceserverroot*\help\toolbox\bugfinder\examples\coding_standards_XML and turn off rules using entries in the XML file (all rules from a standard are enabled in the template). Here, *polyspaceserverroot* is the root installation folder for the Polyspace Server products, for instance, C:\Program Files\Polyspace Server\R2023a.

For instance, to turn off MISRA C:2012 rule 8.1, use this entry in a copy of the file *misra_c_2012_rules.xml*:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
    ...
    <check id="8.1" state="off">
      </check>
    ...
  </section>
  ...
</standard>
```

When using the Guideline checkers, specify their threshold between the `threshold` tags. For instance, to activate the checker `Cyclomatic complexity exceeds threshold` and set the threshold for the checker to five, use this entry in a copy of the *guidelines.xml*:

```
<check id="SC18" state="on">
  <threshold>5</threshold>
</check>
```

To use the XML file for a MISRA C:2012 analysis in Bug Finder, enter:

```
polyspace-bug-finder -sources filename -misra3 from-file
                    -checkers-selection-file misra_c_2012_rules.xml
```

For full list of rule id-s and section names, see:

- "AUTOSAR C++14 Rules"
- "CERT C Rules and Recommendations"
- "CERT C++ Rules"
- "ISO/IEC TS 17961 Rules"
- "Custom Coding Rules"
- "Common Weakness Enumeration (CWE)"
- "JSF C++ Rules"
- "MISRA C:2004 Rules"
- "MISRA C:2012 Directives and Rules"
- "MISRA C++:2008 Rules"
- "Guidelines"

Note The XML format of the checker configuration file can change in future releases.

Check for Coding Standards Only

A Bug Finder analysis checks C/C++ code for:

- A default set of defects (bugs)
- Adherence to any coding standard that you specify.

To check for coding standards only, disable defect checking entirely. Specify none for the option Find defects (-checkers).

Review Coding Rule Violations

The screenshot displays the 'Result Details' pane with the following information:

- Status:** To fix
- Severity:** Medium
- Violation:** MISRA C:2012 5.1 (Required). External identifiers shall be distinct. External function demo_corrected_sighandlerasynccunsafestrict conflicts with the external identifier demo_corrected_sighandlerasynccunsafe (programming.c line 1171).

	Event	File	Scope	Line
1	Violation site	programming.c	programming.c	1171
2	MISRA C:2012 5.1	programming.c	File Scope	1230


The 'Source' pane shows the following code snippet:

```

1226 void corrected_sighandlerasynccunsafestrict(int signum) {
1227     int s0 = signum; /* Fix: avoid raise() */
1228 }
1229
1230 int demo_corrected_sighandlerasynccunsafestrict(void) {
1231     if (signal(SIGTERM, demo_term_handler) == SIG_ERR) {
1232         /* Handle error */
1233     }
1234     if (signal(SIGINT, corrected_sighandlerasynccunsafestrict) == SIG_ERR) {
1235         /* Handle error */
1236     }
1237     /* Program code */
1238     if (raise(SIGINT) != 0) {
1239         /* Handle error */
1240     }
1241     /* More code */
1242     return 0;
1243 }

```

After analysis, you see the coding standard violations on the **Results List** pane. Select a violation to see further details on the **Result Details** pane and the source code on the **Source** pane.

Violations of coding standards are indicated in the source code with the  icon.

For further steps, see “Review Analysis Results” or “Review Polyspace Bug Finder Results in Web Browser”.

Generate Reports

You can generate reports using templates that are explicitly defined for coding standards. Use the `CodingStandards` template. This template:

- Reports only coding standard violations in your analysis results, and omits other types of results such as defects, run-time errors or code metrics.
- Creates a separate chapter in the report for each coding standard. the chapter provides an overview of all violations of the standard and then lists each violation.

To specify a report template, use the option `Bug Finder` and `Code Prover` report (`-report-template`).

See Also

More About

- “Interpret Bug Finder Results in Polyspace Desktop User Interface” on page 21-2
- “Interpret Bug Finder Results in Polyspace Access Web Interface” on page 25-2
- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 22-2
- “Filter and Group Results in Polyspace Desktop User Interface” on page 23-2
- “Generate Reports from Polyspace Results” on page 24-2

Avoid Violations of MISRA C:2012 Rules 8.x

MISRA C:2012 rules 8.1-8.14 enforce good coding practices surrounding declarations and definitions. If you follow these practices, you are less likely to have conflicting declarations or to unintentionally modify variables.

If you do not follow these practices *during coding*, your code might require major changes later to be MISRA C-compliant. You might have too many MISRA C violations. Sometimes, in fixing a violation, you might violate another rule. Instead, keep these rules in mind when coding. Use the MISRA C:2012 checker to spot any issues that you might have missed.

- **Explicitly specify all data types in declarations.**

Avoid implicit data types like this declaration of `k`:

```
extern void foo (char c, const k);
```

Instead use:

```
extern void foo (char c, const int k);
```

That way, you do not violate MISRA C:2012 Rule 8.1.

- **When declaring functions, provide names and data types for all parameters.**

Avoid declarations without parameter names like these declarations:

```
extern int func(int);
extern int func2();
```

Instead use:

```
extern int func(int arg);
extern int func2(void);
```

That way, you do not violate MISRA C:2012 Rule 8.2.

- **If you want to use an object or function in multiple files, declare the object or function once in only one header file.**

To use an object in multiple source files, declare it as `extern` in a header file. Include the header file in all the source files where you need the object. In one of those source files, define the object. For instance:

```
/* header.h */
extern int var;

/* file1.c */
#include "header.h"
/* Some usage of var */

/* file2.c */
#include "header.h"
int var=1;
```

To use a function in multiple source files, declare it in a header file. Include the header file in all the source files where you need the function. In one of those source files, define the function.

That way, you do not violate MISRA C:2012 Rule 8.3, MISRA C:2012 Rule 8.4, MISRA C:2012 Rule 8.5, or MISRA C:2012 Rule 8.6.

- **If you want to use an object or function in one file only, declare and define the object or function with the static specifier.**

Make sure that you use the static specifier in all declarations and the definition. For instance, this function func is meant to be used only in the current file:

```
static int func(void);
static int func(void){
}
```

That way, you do not violate MISRA C:2012 Rule 8.7 and MISRA C:2012 Rule 8.8.

- **If you want to use an object in one function only, declare the object in the function body.**

Avoid declaring the object outside the function.

For instance, if you use var in func only, do declare it outside the body of func:

```
int var;
void func(void) {
    var=1;
}
```

Instead use:

```
void func(void) {
    int var;
    var=1;
}
```

That way, you do not violate MISRA C:2012 Rule 8.7 and MISRA C:2012 Rule 8.9.

- **If you want to inline a function, declare and define the function with the static specifier.**

Every time you add inline to a function definition, add static too:

```
static inline double func(int val);
static inline double func(int val) {
}
```

That way, you do not violate MISRA C:2012 Rule 8.10.

- **When declaring arrays, explicitly specify their size.**

Avoid implicit size specifications like this:

```
extern int32_t array[];
```

Instead use:

```
#define MAXSIZE 10
extern int32_t array[MAXSIZE];
```

That way, you do not violate MISRA C:2012 Rule 8.11.

- **When declaring enumerations, try to avoid mixing implicit and explicit specifications.**

Avoid mixing implicit and explicit specifications. You can specify the first enumeration constant explicitly, but after that, use either implicit or explicit specifications. For instance, avoid this type of mix:

```
enum color {red = 2, blue, green = 3, yellow};
```

Instead use:

```
enum color {red = 2, blue, green, yellow};
```

That way, you do not violate MISRA C:2012 Rule 8.12.

- **When declaring pointers, point to a const-qualified type unless you want to use the pointer to modify an object.**

Point to a const-qualified type by default unless you intend to use the pointer for modifying the pointed object. For instance, in this example, `ptr` is not used to modify the pointed object:

```
char last_char(const char * const ptr){  
}
```

That way, you do not violate MISRA C:2012 Rule 8.13.

Reduce Software Complexity by Using Polyspace Checkers

Software complexity refers to various quantifiable metrics of a software module or source files, such as number of lines, number of paths, number of functions, or the complexity of the function call tree. The Polyspace software complexity checkers are raised when these metrics exceeds a threshold. High software complexity might indicate that your code is difficult to read, understand, and debug. It is more efficient to maintain the acceptable level of software complexity during development instead of refactoring complex projects later on. Use the software complexity checkers to detect complex modules early in the development cycle to reduce later refactoring efforts.

You can also calculate the absolute values of code complexity metrics for all files and functions. See “Compute Code Complexity Metrics Using Polyspace” on page 16-51.

Configure Thresholds for Software Complexity Checkers

Each software complexity checker corresponds to a complexity metric. Polyspace raises a software complexity checker when the corresponding code complexity metric exceeds a threshold.

The default thresholds of these checkers follow the Hersteller Initiative Software (HIS) Code Complexity standard. See “HIS Code Complexity Metrics” on page 16-54. For checkers that are not present in the HIS standard, the default thresholds are high enough that the code complexity metrics of your code might always be below the threshold. To use these checkers effectively, specify an appropriate threshold for them.

Determine an appropriate set of thresholds for these checkers depending on the best practice for your use case. For instance, when analyzing new projects or newly developed code, you might want to reduce the use of GOTO statements by setting the threshold of `Number of goto statements exceeds threshold` to zero. When analyzing modules containing legacy libraries, you might want to set the threshold to a higher number.

Depending on your Polyspace product, use the user interface or the command-line interface to specify the threshold. For instance:

- In Polyspace desktop or Server products, in the Checkers selection window, navigate to **Guidelines > Software Complexity** and specify the threshold. In the command line, use the analysis option `Check guidelines (-guidelines)`. See “Check for and Review Coding Standard Violations” on page 16-2.
- In Polyspace as You Code extension, start the Checkers selection window and specify the thresholds in the **Guidelines > Software Complexity** node.
 - In Eclipse, open the Checkers selection window from the Configure Project window. See “Configure Checkers for Polyspace as You Code in Eclipse” on page 11-60.
 - In Visual Studio, open the Checkers selection window from the **Polyspace > Project** node of the Options window. See “Configure Checkers for Polyspace as You Code in Visual Studio” on page 11-63.
 - In Visual Studio Code, open the Checkers selection window from the command palette. See “Configure Checkers for Polyspace as You Code in Visual Studio Code” on page 11-66.
 - At the command line, open the Checkers selection window by running the command `polyspace-checkers-selection`. See “Configure Checkers for Polyspace as You Code at the Command Line” on page 11-70.

Identify and Reduce Software Complexity

Identify Software Complexity by Running Bug Finder Analysis

To identify software complexity, configure the thresholds of the checkers. For instance, set the thresholds of the checkers listed in this table.

Checker	Threshold
Comment density below threshold	20
Call tree complexity exceeds threshold	10
Number of call occurrences exceeds threshold	10
Language scope exceeds threshold	400

The thresholds indicate the acceptable level of software complexity. To identify issues in your code that might lead to a higher level of complexity, after configuring the software complexity checkers, run a Polyspace Bug Finder analysis. Consider this code:

```
long long power(double x, int n){
    long long BN = 1;
    for(int i = 0; i<n;++i){
        BN*=x;
    }
    return BN;
}

double AppxIndex(double m, double f){//Noncompliant
    double U = (power(m,2) - 1)/(power(m,2)+2);
    double V = (power(m,4) + 27*power(m,2)+38)/(2*power(m,2)+3);
    return (1+2*f*power(U,2)*(1+power(m,2)*U*V+ power(m,3)
        /power(m,3)*(U-V)))/( (1-2*f*power(U,2)*(1+power(m,2)*U*V
        + power(m,3)/power(m,3)*(U-V))));
}
```

The function `AppxIndex` appears complex. It is not obvious how you might reduce the complexity. The software complexity checkers help you identify the sources of complexity.

After the Bug Finder analysis, the configured checkers are raised:

- **Comment density below threshold:** The functions in the code contain no explanatory comments.
- **Call tree complexity exceeds threshold and Number of call occurrences exceeds threshold:** There are too many function calls compared to the number of function definitions. These checks indicate that you can package some of the expressions into separate functions.
- **Language scope exceeds threshold:** The same operand is repeated several times. You can reduce some of the repetition. For instance, the function `power` is called with the same arguments several times.

These checks indicate that the function `AppxIndex` might make the code difficult to read, understand, and debug. To reduce the complexity of the code, address the raised checks.

Reduce Software Complexity

Reduce the complexity of your code by addressing the identified issues. In this case, the root cause of the raised checks is that the function `AppxIndex` performs several tasks instead of performing one single task. For instance, the function first calculates `U`, then it calculates `V`, and finally it evaluates a lengthy expression containing both `U` and `V`. To address these issues, refactor the function `AppxIndex` so that each task is delegated to a separate function. You might break down the lengthy expression into smaller parts. For instance:

```
// This code calculates effective index of materials as described in
// the formula in 10.1364...
// power(x,n) returns the nth power of x (x^n)
// n is an integer
// x is a double
// return type is long long

long long power(double x, int n){//Compliant
    long long BN = 1;
    for(int i = 0; i<n;++i){
        BN*=x;
    }
    return BN;
}
// CalculateU(m) calculates the first intermediate variable
// required to calculate polarization
// m is the relative refractive index
// return type is double;

double CalculateU(double m){//Compliant
    return (power(m,2) - 1)/(power(m,2)+2);
}
// CalculateV(m) calculates the second intermediate variable
// required to calculate polarization
// m is the relative refractive index
// return type is double;

double CalculateV(double m){//Compliant
    return (power(m,4) + 27*power(m,2)+38)/(2*power(m,2)+3);
}
// CalculateMid(m,f) calculates the large term present
// in both numerator and denominator
// of the effective index calculation
// m is the relative refractive index
// f is the fillfactor
// return type is double;

double CalculateMid(double m, double f){//Compliant
    double U = CalculateU(m);
    double V = CalculateU(m);
    return 2*f*power(U,2)*(1+power(m,2)*U*V + power(m,3)/power(m,3)*(U-V));
}
//AppxIndex(m,f) calculates the approximate effective index
// m is the relative refractive index
// f is the fillfactor
//return type is double
double AppxIndex(double m, double f){//Compliant
```

```
    return (1+CalculateMid(m,f))/( (1-CalculateMid(m,f)));  
}
```

In this code, none of the software complexity checkers is raised, which indicates that you reduced the complexity of this code to an acceptable level. To reduce the software complexity:

- 1** Document the code with sufficient comments.
- 2** Break down the The large complex task performed by `AppxIndex` into smaller and simpler tasks, which are then delegated to individual functions such as `CalculateU`, `CalculateV` and `CalculateMid`. The function `power` is now called less frequently. If you later implement a different function to calculate a power and want to use the new function instead of the current one, you have to make fewer replacements.
- 3** Write the new functions to perform one specific task with as little overlap of their functionalities as possible. As a result, these functions contain less repetition of the same operands.

For details about addressing a software complexity check, see the documentation of the checker.

In cases when you are unable to refactor the code, address the checks through code annotations. For instance, if you are using a complex library, you might choose to annotate the checks that are raised on the library. See “Annotate Code and Hide Known or Acceptable Results” on page 30-2. When you annotate a file or function code metric, the corresponding software complexity checker is also annotated by the same comment.

See Also

More About

- “Guidelines”

Software Quality Objective Subsets (C:2004)

In this section...
“Rules in SQO-Subset1” on page 16-15
“Rules in SQO-Subset2” on page 16-16

Rules in SQO-Subset1

The SQO subset1 consists of these rules:

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
8.11	The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
12.12	The underlying bit representations of floating-point values shall not be used.
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a for statement shall not contain any objects of floating type.
13.5	The three expressions of a for statement shall be concerned only with loop control.
14.4	The goto statement shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	The declaration of objects should contain no more than 2 levels of pointer indirection.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.4	Unions shall not be used.
20.4	Dynamic heap memory allocation shall not be used.

Rules in SQO-Subset2

Good design practices generally lead to less code complexity. The following set of coding rules enforce good design practices. The SQO-subset2 option checks the rules in SQO-subset1 and some additional rules.

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
6.3	Typedefs that indicate size and signedness should be used in place of the basic types.
8.7	Objects shall be defined at block scope if they are only accessed from within a single function.
8.11	The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation.
9.2	Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures.
9.3	In an enumerator list, the '=' construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly initialised.
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.
10.5	If the bitwise operator ~ and && are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
11.5	A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.
12.1	Limited dependence should be placed on C's operator precedence rules in expressions.
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits.
12.5	The operands of a logical && or shall be primary-expressions.
12.6	The operands of a logical operators (&&, and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, and !).
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Rule number	Description
12.10	The comma operator shall not be used.
12.12	The underlying bit representations of floating-point values shall not be used.
13.1	Assignment operators shall not be used in expressions that yield a Boolean value.
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a for statement shall not contain any objects of floating type.
13.5	The three expressions of a for statement shall be concerned only with loop control.
13.6	Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop.
14.4	The goto statement shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
14.8	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.
14.10	All if ... else if constructs should contain a final else clause.
15.3	The final clause of a switch statement shall be the default clause.
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.
16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesised parameter list, which may be empty.
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	The declaration of objects should contain no more than 2 levels of pointer indirection.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.4	Unions shall not be used.
19.4	C macros shall only expand to a braced initialiser, a constant, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct.

Rule number	Description
19.9	Arguments to a function-like macro shall not contain tokens that look like pre-processing directives.
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.
19.12	There shall be at most one occurrence of the # or ## pre-processor operators in a single macro definition.
20.3	The validity of values passed to library functions shall be checked.
20.4	Dynamic heap memory allocation shall not be used.

Note Polyspace software does not check MISRA rule **20.3** directly.

However, you can check this rule by writing manual stubs that check the validity of values. For example, the following code checks the validity of an input being greater than 1:

```
int my_system_library_call(int in) {assert (in>1); if random \  
return -1 else return 0; }
```

See Also

Check MISRA C:2004 (-misra2)

More About

- “Check for and Review Coding Standard Violations” on page 16-2

Software Quality Objective Subsets (AC AGC)

In this section...
“Rules in SQO-Subset1” on page 16-19
“Rules in SQO-Subset2” on page 16-20

Rules in SQO-Subset1

The SQO subset1 consists of these rules:

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier
8.11	The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to object type or a pointer to void
11.3	A cast should not be performed between a pointer type and an integral type
12.12	The underlying bit representations of floating-point values shall not be used
13.4	The controlling expression of a for statement shall not contain any objects of floating type
13.5	The three expressions of a for statement shall be concerned only with loop control
14.4	The goto statement shall not be used
14.7	A function shall have a single point of exit at the end of the function
16.1	Functions shall not be defined with variable numbers of arguments
16.2	Functions shall not call themselves, either directly or indirectly
16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array
17.4	Array indexing shall be the only allowed form of pointer arithmetic
17.5	The declaration of objects should contain no more than 2 levels of pointer indirection
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist
18.4	Unions shall not be used
20.4	Dynamic heap memory allocation shall not be used

For more information about these rules, see *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

Rules in SQO-Subset2

Good design practices generally lead to less code complexity. The following set of coding rules enforce good design practices. The `SQO-subset2` option checks the rules in `SQO-subset1` and some additional rules.

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier
6.3	Typedefs that indicate size and signedness should be used in place of the basic types
8.7	Objects shall be defined at block scope if they are only accessed from within a single function
8.11	The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation
9.2	Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures
9.3	In an enumerator list, the '=' construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly initialised
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression
10.5	If the bitwise operator <code>~</code> and <code><<</code> are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to object type or a pointer to void
11.3	A cast should not be performed between a pointer type and an integral type
11.5	A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer
12.1	Limited dependence should be placed on C's operator precedence rules in expressions
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits
12.5	The operands of a logical <code>&&</code> or <code> </code> shall be primary-expressions
12.6	The operands of a logical operators (<code>&&</code> , <code> </code> and <code>!</code>) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (<code>&&</code> , <code> </code> and <code>!</code>)
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned

Rule number	Description
12.10	The comma operator shall not be used
12.12	The underlying bit representations of floating-point values shall not be used
13.1	Assignment operators shall not be used in expressions that yield a Boolean value
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean
13.4	The controlling expression of a for statement shall not contain any objects of floating type
13.5	The three expressions of a for statement shall be concerned only with loop control
13.6	Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop
14.4	The goto statement shall not be used
14.7	A function shall have a single point of exit at the end of the function
14.8	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement
14.10	All if ... else if constructs should contain a final else clause
16.1	Functions shall not be defined with variable numbers of arguments
16.2	Functions shall not call themselves, either directly or indirectly
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration
16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesised parameter list, which may be empty
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array
17.4	Array indexing shall be the only allowed form of pointer arithmetic
17.5	The declaration of objects should contain no more than 2 levels of pointer indirection
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist
18.4	Unions shall not be used
19.4	C macros shall only expand to a braced initialiser, a constant, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct
19.9	Arguments to a function-like macro shall not contain tokens that look like pre-processing directives

Rule number	Description
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator
19.12	There shall be at most one occurrence of the # or ## pre-processor operators in a single macro definition
20.3	The validity of values passed to library functions shall be checked
20.4	Dynamic heap memory allocation shall not be used

Note Polyspace software does not check MISRA rule **20.3** directly.

However, you can check this rule by writing manual stubs that check the validity of values. For example, the following code checks the validity of an input being greater than 1:

```
int my_system_library_call(int in) {assert (in>1); if random \  
return -1 else return 0; }
```

For more information about these rules, see *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

See Also

Check MISRA AC AGC (-misra-ac-agc)

More About

- “Check for and Review Coding Standard Violations” on page 16-2

Software Quality Objective Subsets (C:2012)

In this section...

“Guidelines in SQO-Subset1” on page 16-23

“Guidelines in SQO-Subset2” on page 16-24

These subsets of MISRA C:2012 guidelines can have a direct or indirect impact on the precision of your Polyspace results. When you set up coding rules checking, you can select these subsets.

Guidelines in SQO-Subset1

The SQO subset1 consists of these rules:

Rule	Description
D1.1	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood.
5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.
8.11	When an array with external linkage is declared, its size should be explicitly specified.
8.13	A pointer should point to a const-qualified type whenever possible.
11.1	Conversions shall not be performed between a pointer to a function and any other type.
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type.
11.4	A conversion should not be performed between a pointer to object and an integer type.
11.5	A conversion should not be performed from pointer to void into pointer to object.
11.6	A cast shall not be performed between pointer to void and an arithmetic type.
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type.
14.1	A loop counter shall not have essentially floating type.
14.2	A for loop shall be well-formed.
15.1	The goto statement should not be used.
15.2	The goto statement shall jump to a label declared later in the same function.
15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.
15.5	A function should have a single point of exit at the end.
17.1	The features of <stdarg.h> shall not be used.
17.2	Functions shall not call themselves, either directly or indirectly.

Rule	Description
18.3	The relational operators <code>></code> , <code>>=</code> , <code><</code> and <code><=</code> shall not be applied to objects of pointer type except where they point into the same object.
18.4	The <code>+</code> , <code>-</code> , <code>+=</code> and <code>-=</code> operators should not be applied to an expression of pointer type.
18.5	Declarations should contain no more than two levels of pointer nesting.
18.6	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.
19.2	The union keyword should not be used.
21.3	The memory allocation and deallocation functions of <code><stdlib.h></code> shall not be used.

Guidelines in SQO-Subset2

Good design practices generally lead to less code complexity. The following set of coding rules enforce good design practices. The `SQO-subset2` option checks the rules in `SQO-subset1` and some additional rules.

Rule	Description
D1.1	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood.
D4.6	typedefs that indicate size and signedness should be used in place of the basic numerical types.
D4.11	The validity of values passed to library functions shall be checked.
5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
8.2	Function types shall be in prototype form with named parameters.
8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.
8.9	An object should be defined at block scope if its identifier only appears in a single function.
8.11	When an array with external linkage is declared, its size should be explicitly specified.
8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.
8.13	A pointer should point to a const-qualified type whenever possible.
9.2	The initializer for an aggregate or union shall be enclosed in braces.
9.3	Arrays shall not be partially initialized.
10.1	Operands shall not be of an inappropriate essential type.
10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type.
11.1	Conversions shall not be performed between a pointer to a function and any other type.

Rule	Description
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type.
11.4	A conversion should not be performed between a pointer to object and an integer type.
11.5	A conversion should not be performed from pointer to void into pointer to object.
11.6	A cast shall not be performed between pointer to void and an arithmetic type.
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type.
11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.
12.1	The precedence of operators within expressions should be made explicit.
12.3	The comma operator should not be used
13.2	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.
13.4	The result of an assignment operator should not be used.
14.1	A loop counter shall not have essentially floating type.
14.2	A for loop shall be well-formed.
14.4	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type.
15.1	The goto statement should not be used.
15.2	The goto statement shall jump to a label declared later in the same function.
15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.
15.5	A function should have a single point of exit at the end.
15.6	The body of an iteration-statement or a selection-statement shall be a compound-statement.
15.7	All if ... else if constructs shall be terminated with an else statement.
16.4	Every switch statement shall have a default label.
16.5	A default label shall appear as either the first or the last switch label of a switch statement.
17.1	The features of <stdarg.h> shall not be used.
17.2	Functions shall not call themselves, either directly or indirectly.
17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
18.3	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object.
18.4	The +, -, += and -= operators should not be applied to an expression of pointer type.
18.5	Declarations should contain no more than two levels of pointer nesting.

Rule	Description
18.6	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.
19.2	The union keyword should not be used.
20.4	A macro shall not be defined with the same name as a keyword.
20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument.
20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.
20.9	All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation.
20.11	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.
21.3	The memory allocation and deallocation functions of <stdlib.h> shall not be used.

See Also

Check MISRA C:2012 (-misra3)

More About

- “Check for and Review Coding Standard Violations” on page 16-2

Software Quality Objective Subsets (C++)

In this section...

“SQO Subset 1 - Direct Impact on Selectivity” on page 16-27

“SQO Subset 2 - Indirect Impact on Selectivity” on page 16-28

SQO Subset 1 - Direct Impact on Selectivity

The SQO subset 1 consists of these MISRA C++:2008 rules:

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.
3-9-3	The underlying bit representations of floating-point values shall not be used.
5-0-15	Array indexing shall be the only form of pointer arithmetic.
5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.
6-6-5	A function shall have a single point of exit at the end of the function.
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

MISRA C++ Rule	Description
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.
7-5-4	Functions should not call themselves, either directly or indirectly.
8-4-1	Functions shall not be defined using the ellipsis notation.
9-5-1	Unions shall not be used.
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.
10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy.
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base clas s.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
18-4-1	Dynamic heap memory allocation shall not be used.

SQO Subset 2 - Indirect Impact on Selectivity

Good design practices generally lead to less code complexity. The following set of coding rules may help to address design issues in your code. The SQO-subset2 option checks the rules in SQO-subset1 and SQO-subset2.

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

MISRA C++ Rule	Description
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.
3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.
3-9-2	Typedefs that indicate size and signedness should be used in place of the basic numerical types.
3-9-3	The underlying bit representations of floating-point values shall not be used.
4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, , !, the equality operators == and !=, the unary & operator, and the conditional operator.
5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.
5-0-2	Limited dependence should be placed on C++ operator precedence rules in expressions.
5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression.
5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.
5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.
5-0-10	If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.
5-0-13	The condition of an if-statement and the condition of an iteration-statement shall have type bool.
5-0-15	Array indexing shall be the only form of pointer arithmetic.
5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.
5-2-1	Each operand of a logical && or shall be a postfix-expression.
5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.
5-2-5	A cast shall not remove any const or volatile qualification from the type of a pointer or reference.
5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.
5-2-7	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.
5-2-11	The comma operator, && operator and the operator shall not be overloaded.

MISRA C++ Rule	Description
5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
5-3-3	The unary & operator shall not be overloaded.
5-18-1	The comma operator shall not be used.
6-2-1	Assignment operators shall not be used in sub-expressions.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.
6-4-2	All if ... else if constructs shall be terminated with an else clause.
6-4-6	The final clause of a switch statement shall be the default-clause.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.
6-6-5	A function shall have a single point of exit at the end of the function.
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.
7-5-4	Functions should not call themselves, either directly or indirectly.
8-4-1	Functions shall not be defined using the ellipsis notation.
8-4-3	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &.
8-5-2	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.
8-5-3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.
9-5-1	Unions shall not be used.
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.
10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy.

MISRA C++ Rule	Description
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.
11-0-1	Member data in non-POD class types shall be private.
12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor.
12-8-2	The copy assignment operator shall be declared protected or private in an abstract class.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.
16-0-7	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.
16-2-2	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.
16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition.
18-4-1	Dynamic heap memory allocation shall not be used.

See Also

Check MISRA C++:2008 (-misra-cpp)

More About

- “Check for and Review Coding Standard Violations” on page 16-2

Coding Rule Subsets Checked Early in Analysis

In the initial compilation phase of the analysis, Polyspace checks those coding rules that do not require the run-time error detection part of the analysis. If you want only those rules checked, you can perform a much quicker analysis.

The software provides two predefined subsets of rules that it checks earlier in the analysis. The subsets are available with the options Check MISRA C:2004 (-misra2), Check MISRA AC AGC (-misra-ac-agc), and Check MISRA C:2012 (-misra3).

Argument	Purpose
single-unit-rules	<p>Check rules that apply only to single translation units.</p> <p>If you detect only coding rule violations and select this subset, a Bug Finder analysis stops after the compilation phase.</p>
system-decidable-rules	<p>Check rules in the single-unit-rules subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit.</p> <p>If you detect only coding rule violations and select this subset, a Bug Finder analysis stops after the linking phase.</p>

See also “Check for and Review Coding Standard Violations” on page 16-2.

MISRA C:2004 and MISRA AC AGC Rules

The software checks the following rules early in the analysis. The rules that are checked at a system level and appear only in the system-decidable-rules subset are indicated by an asterisk.

Environment

Rule	Description
1.1*	All code shall conform to ISO 9899:1990 "Programming languages - C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.

Language Extensions

Rule	Description
2.1	Assembly language shall be encapsulated and isolated.
2.2	Source code shall only use /* */ style comments.
2.3	The character sequence /* shall not be used within a comment.

Documentation

Rule	Description
3.4	All uses of the #pragma directive shall be documented and explained.

Character Sets

Rule	Description
4.1	Only those escape sequences which are defined in the ISO C standard shall be used.
4.2	Trigraphs shall not be used.

Identifiers

Rule	Description
5.1*	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
5.3*	A typedef name shall be a unique identifier.
5.4*	A tag name shall be a unique identifier.
5.5*	No object or function identifier with a static storage duration should be reused.
5.6*	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.
5.7*	No identifier name should be reused.

Types

Rule	Description
6.1	The plain char type shall be used only for the storage and use of character values.
6.2	Signed and unsigned char type shall be used only for the storage and use of numeric values.
6.3	typedefs that indicate size and signedness should be used in place of the basic types.
6.4	Bit fields shall only be defined to be of type <code>unsigned int</code> or <code>signed int</code> .
6.5	Bit fields of type <code>signed int</code> shall be at least 2 bits long.

Constants

Rule	Description
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and Definitions

Rule	Description
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated.
8.3	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.
8.4*	If objects or functions are declared more than once their types shall be compatible.
8.5	There shall be no definitions of objects or functions in a header file.
8.6	Functions shall always be declared at file scope.
8.7	Objects shall be defined at block scope if they are only accessed from within a single function.
8.8*	An external object or function shall be declared in one file and only one file.
8.9*	An identifier with external linkage shall have exactly one external definition.
8.10*	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.
8.11	The <code>static</code> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

Initialization

Rule	Description
9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.
9.3	In an enumerator list, the <code>=</code> construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Arithmetic Type Conversion

Rule	Description
10.1	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: <ul style="list-style-type: none"> • It is not a conversion to a wider integer type of the same signedness, or • The expression is complex, or • The expression is not constant and is a function argument, or • The expression is not constant and is a return expression
10.2	The value of an expression of floating type shall not be implicitly converted to a different type if <ul style="list-style-type: none"> • It is not a conversion to a wider floating type, or • The expression is complex, or • The expression is a function argument, or • The expression is a return expression
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.
10.4	The value of a complex expression of float type may only be cast to narrower floating type.
10.5	If the bitwise operator <code>~</code> and <code><<</code> are applied to an operand of underlying type <code>unsigned char</code> or <code>unsigned short</code> , the result shall be immediately cast to the underlying type of the operand
10.6	The "U" suffix shall be applied to all constants of <code>unsigned</code> types.

Pointer Type Conversion

Rule	Description
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to <code>void</code> .
11.3	A cast should not be performed between a pointer type and an integral type.
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.
11.5	A cast shall not be performed that removes any <code>const</code> or <code>volatile</code> qualification from the type addressed by a pointer

Expressions

Rule	Description
12.1	Limited dependence should be placed on C's operator precedence rules in expressions.
12.3	The <code>sizeof</code> operator should not be used on expressions that contain side effects.
12.5	The operands of a logical <code>&&</code> or <code> </code> shall be primary-expressions.
12.6	Operands of logical operators (<code>&&</code> , <code> </code> and <code>!</code>) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (<code>&&</code> , <code> </code> or <code>!</code>).
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed.
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
12.10	The comma operator shall not be used.
12.11	Evaluation of constant unsigned expression should not lead to wraparound.
12.12	The underlying bit representations of floating-point values shall not be used.
12.13	The increment (<code>++</code>) and decrement (<code>--</code>) operators should not be mixed with other operators in an expression

Control Statement Expressions

Rule	Description
13.1	Assignment operators shall not be used in expressions that yield Boolean values.
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a <code>for</code> statement shall not contain any objects of floating type.
13.5	The three expressions of a <code>for</code> statement shall be concerned only with loop control.
13.6	Numeric variables being used within a <code>for</code> loop for iteration counting should not be modified in the body of the loop.

Control Flow

Rule	Description
14.3	All non-null statements shall either <ul style="list-style-type: none"> • have at least one side effect however executed, or • cause control flow to change.
14.4	The <code>goto</code> statement shall not be used.
14.5	The <code>continue</code> statement shall not be used.
14.6	For any iteration statement, there shall be at most one <code>break</code> statement used for loop termination.
14.7	A function shall have a single point of exit at the end of the function.
14.8	The statement forming the body of a <code>switch</code> , <code>while</code> , <code>do while</code> or <code>for</code> statement shall be a compound statement.
14.9	An <code>if (expression)</code> construct shall be followed by a compound statement. The <code>else</code> keyword shall be followed by either a compound statement, or another <code>if</code> statement.
14.10	All <code>if else if</code> constructs should contain a final <code>else</code> clause.

Switch Statements

Rule	Description
15.0	Unreachable code is detected between <code>switch</code> statement and first <code>case</code> .
15.1	A <code>switch</code> label shall only be used when the most closely-enclosing compound statement is the body of a <code>switch</code> statement
15.2	An unconditional <code>break</code> statement shall terminate every non-empty <code>switch</code> clause.
15.3	The final clause of a <code>switch</code> statement shall be the <code>default</code> clause.
15.4	A <code>switch</code> expression should not represent a value that is effectively Boolean.
15.5	Every <code>switch</code> statement shall have at least one <code>case</code> clause.

Functions

Rule	Description
16.1	Functions shall not be defined with variable numbers of arguments.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.
16.4*	The identifiers used in the declaration and definition of a function shall be identical.
16.5	Functions with no parameters shall be declared with parameter type <code>void</code> .
16.6	The number of arguments passed to a function shall match the number of parameters.
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
16.9	A function identifier shall only be used with either a preceding <code>&</code> , or with a parenthesized parameter list, which may be empty.

Pointers and Arrays

Rule	Description
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	A type should not contain more than 2 levels of pointer indirection.

Structures and Unions

Rule	Description
18.1	All structure or union types shall be complete at the end of a translation unit.
18.4	Unions shall not be used.

Preprocessing Directives

Rule	Description
19.1	<code>#include</code> statements in a file shall only be preceded by other preprocessors directives or comments.
19.2	Nonstandard characters should not occur in header file names in <code>#include</code> directives.
19.3	The <code>#include</code> directive shall be followed by either a <code><filename></code> or "filename" sequence.
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.
19.5	Macros shall not be <code>#defined</code> and <code>#undefd</code> within a block.
19.6	<code>#undef</code> shall not be used.
19.7	A function should be used in preference to a function like-macro.
19.8	A function-like macro shall not be invoked without all of its arguments.
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
19.10	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of <code>#</code> or <code>##</code> .
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in <code>#ifdef</code> and <code>#ifndef</code> preprocessor directives and the <code>defined()</code> operator.
19.12	There shall be at most one occurrence of the <code>#</code> or <code>##</code> preprocessor operators in a single macro definition.
19.13	The <code>#</code> and <code>##</code> preprocessor operators should not be used.
19.14	The <code>defined</code> preprocessor operator shall only be used in one of the two standard forms.
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.
19.17	All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> or <code>#ifdef</code> directive to which they are related.

Standard Libraries

Rule	Description
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.
20.2	The names of standard library macros, objects and functions shall not be reused.
20.4	Dynamic heap memory allocation shall not be used.
20.5	The error indicator <code>errno</code> shall not be used.
20.6	The macro <code>offsetof</code> , in library <code><stddef.h></code> , shall not be used.
20.7	The <code>setjmp</code> macro and the <code>longjmp</code> function shall not be used.
20.8	The signal handling facilities of <code><signal.h></code> shall not be used.
20.9	The input/output library <code><stdio.h></code> shall not be used in production code.
20.10	The library functions <code>atof</code> , <code>atoi</code> and <code>atoll</code> from library <code><stdlib.h></code> shall not be used.
20.11	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> from library <code><stdlib.h></code> shall not be used.
20.12	The time handling functions of library <code><time.h></code> shall not be used.

The rules that are checked at a system level and appear only in the `system-decidable-rules` subset are indicated by an asterisk.

MISRA C:2012 Rules

The software checks the following rules early in the analysis. The rules that are checked at a system level and appear only in the `system-decidable-rules` subset are indicated by an asterisk.

Standard C Environment

Rule	Description
1.1	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.
1.2	Language extensions should not be used.

Unused Code

Rule	Description
2.3*	A project should not contain unused type declarations.
2.4*	A project should not contain unused tag declarations.
2.5*	A project should not contain unused macro declarations.
2.6	A function should not contain unused label declarations.
2.7	There should be no unused parameters in functions.

Comments

Rule	Description
3.1	The character sequences <code>/*</code> and <code>//</code> shall not be used within a comment.
3.2	Line-splicing shall not be used in <code>//</code> comments.

Character Sets and Lexical Conventions

Rule	Description
4.1	Octal and hexadecimal escape sequences shall be terminated.
4.2	Trigraphs should not be used.

Identifiers

Rule	Description
5.1*	External identifiers shall be distinct.
5.2	Identifiers declared in the same scope and name space shall be distinct.
5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
5.4	Macro identifiers shall be distinct.
5.5	Identifiers shall be distinct from macro names.
5.6*	A typedef name shall be a unique identifier.
5.7*	A tag name shall be a unique identifier.
5.8*	Identifiers that define objects or functions with external linkage shall be unique.
5.9*	Identifiers that define objects or functions with internal linkage should be unique.

Types

Rule	Description
6.1	Bit-fields shall only be declared with an appropriate type.
6.2	Single-bit named bit fields shall not be of a signed type.

Literals and Constants

Rule	Description
7.1	Octal constants shall not be used.
7.2	A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type.
7.3	The lowercase character "l" shall not be used in a literal suffix.
7.4	A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

Declarations and Definitions

Rule	Description
8.1	Types shall be explicitly specified.
8.2	Function types shall be in prototype form with named parameters.
8.3*	All declarations of an object or function shall use the same names and type qualifiers.
8.4	A compatible declaration shall be visible when an object or function with external linkage is defined.
8.5*	An external object or function shall be declared once in one and only one file.
8.6*	An identifier with external linkage shall have exactly one external definition.
8.7*	Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.
8.8	The <code>static</code> storage class specifier shall be used in all declarations of objects and functions that have internal linkage.
8.9*	An object should be defined at block scope if its identifier only appears in a single function.
8.10	An inline function shall be declared with the <code>static</code> storage class.
8.11	When an array with external linkage is declared, its size should be explicitly specified.
8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.
8.14	The <code>restrict</code> type qualifier shall not be used.

Initialization

Rule	Description
9.2	The initializer for an aggregate or union shall be enclosed in braces.
9.3	Arrays shall not be partially initialized.
9.4	An element of an object shall not be initialized more than once.
9.5	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

The Essential Type Model

Rule	Description
10.1	Operands shall not be of an inappropriate essential type.
10.2	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.
10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.
10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.
10.5	The value of an expression should not be cast to an inappropriate essential type.
10.6	The value of a composite expression shall not be assigned to an object with wider essential type.
10.7	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.
10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

Pointer Type Conversion

Rule	Description
11.1	Conversions shall not be performed between a pointer to a function and any other type.
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type.
11.3	A cast shall not be performed between a pointer to object type and a pointer to a different object type.
11.4	A conversion should not be performed between a pointer to object and an integer type.
11.5	A conversion should not be performed from pointer to void into pointer to object.
11.6	A cast shall not be performed between pointer to void and an arithmetic type.
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type.
11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.
11.9	The macro NULL shall be the only permitted form of integer null pointer constant.

Expressions

Rule	Description
12.1	The precedence of operators within expressions should be made explicit.
12.3	The comma operator should not be used.
12.4	Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Side Effects

Rule	Description
13.3	A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.
13.4	The result of an assignment operator should not be used.
13.6	The operand of the <code>sizeof</code> operator shall not contain any expression which has potential side effects.

Control Statement Expressions

Rule	Description
14.4	The controlling expression of an <code>if</code> statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

Control Flow

Rule	Description
15.1	The <code>goto</code> statement should not be used.
15.2	The <code>goto</code> statement shall jump to a label declared later in the same function.
15.3	Any label referenced by a <code>goto</code> statement shall be declared in the same block, or in any block enclosing the <code>goto</code> statement.
15.4	There should be no more than one <code>break</code> or <code>goto</code> statement used to terminate any iteration statement.
15.5	A function should have a single point of exit at the end
15.6	The body of an iteration-statement or a selection-statement shall be a compound statement.
15.7	All <code>if ... else if</code> constructs shall be terminated with an <code>else</code> statement.

Switch Statements

Rule	Description
16.1	All <code>switch</code> statements shall be well-formed.
16.2	A <code>switch</code> label shall only be used when the most closely-enclosing compound statement is the body of a <code>switch</code> statement.
16.3	An unconditional <code>break</code> statement shall terminate every <code>switch</code> -clause.
16.4	Every <code>switch</code> statement shall have a <code>default</code> label.
16.5	A <code>default</code> label shall appear as either the first or the last <code>switch</code> label of a <code>switch</code> statement.
16.6	Every <code>switch</code> statement shall have at least two <code>switch</code> -clauses.
16.7	A <code>switch</code> -expression shall not have essentially Boolean type.

Functions

Rule	Description
17.1	The features of <stdarg.h> shall not be used.
17.3	A function shall not be declared implicitly.
17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
17.6	The declaration of an array parameter shall not contain the <code>static</code> keyword between the [].
17.7	The value returned by a function having non-void return type shall be used.

Pointers and Arrays

Rule	Description
18.4	The +, -, += and -= operators should not be applied to an expression of pointer type.
18.5	Declarations should contain no more than two levels of pointer nesting.
18.7	Flexible array members shall not be declared.
18.8	Variable-length array types shall not be used.

Overlapping Storage

Rule	Description
19.2	The <code>union</code> keyword should not be used.

Preprocessing Directives

Rule	Description
20.1	<code>#include</code> directives should only be preceded by preprocessor directives or comments.
20.2	The <code>'</code> , <code>"</code> , or <code>\</code> characters and the <code>/*</code> or <code>//</code> character sequences shall not occur in a header file name.
20.3	The <code>#include</code> directive shall be followed by either a <code><filename></code> or <code>"filename"</code> sequence.
20.4	A macro shall not be defined with the same name as a keyword.
20.5	<code>#undef</code> should not be used.
20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument.
20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.
20.8	The controlling expression of a <code>#if</code> or <code>#elif</code> preprocessing directive shall evaluate to 0 or 1.
20.9	All identifiers used in the controlling expression of <code>#if</code> or <code>#elif</code> preprocessing directives shall be <code>#define</code> 'd before evaluation.
20.10	The <code>#</code> and <code>##</code> preprocessor operators should not be used.
20.11	A macro parameter immediately following a <code>#</code> operator shall not immediately be followed by a <code>##</code> operator.
20.12	A macro parameter used as an operand to the <code>#</code> or <code>##</code> operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.
20.13	A line whose first token is <code>#</code> shall be a valid preprocessing directive.
20.14	All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> , <code>#ifdef</code> or <code>#ifndef</code> directive to which they are related.

Standard Libraries

Rule	Description
21.1	<code>#define</code> and <code>#undef</code> shall not be used on a reserved identifier or reserved macro name.
21.2	A reserved identifier or macro name shall not be declared.
21.3	The memory allocation and deallocation functions of <code><stdlib.h></code> shall not be used.
21.4	The standard header file <code><setjmp.h></code> shall not be used.
21.5	The standard header file <code><signal.h></code> shall not be used.
21.6	The Standard Library input/output functions shall not be used.
21.7	The <code>atof</code> , <code>atoi</code> , <code>atol</code> , and <code>atoll</code> functions of <code><stdlib.h></code> shall not be used.
21.8	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> of <code><stdlib.h></code> shall not be used.
21.9	The library functions <code>bsearch</code> and <code>qsort</code> of <code><stdlib.h></code> shall not be used.
21.10	The Standard Library time and date functions shall not be used.
21.11	The standard header file <code><tgmath.h></code> shall not be used.
21.12	The exception handling features of <code><fenv.h></code> should not be used.

The rules that are checked at a system level and appear only in the `system-decidable-rules` subset are indicated by an asterisk.

See Also

Check MISRA C:2004 (`-misra2`) | Check MISRA AC AGC (`-misra-ac-agc`) | Check MISRA C:2012 (`-misra3`)

More About

- “Check for and Review Coding Standard Violations” on page 16-2

Create Custom Coding Rules

This example shows how to check for violations of naming conventions on functions and objects in your C/C++ code. For each naming convention, you specify a pattern in the form of a regular expression. The software compares the pattern to identifiers in the source code and determines whether the identifiers follow those naming conventions.

The tutorial uses this code stored in a file `printInitialValue.c`:

```
#include <stdio.h>

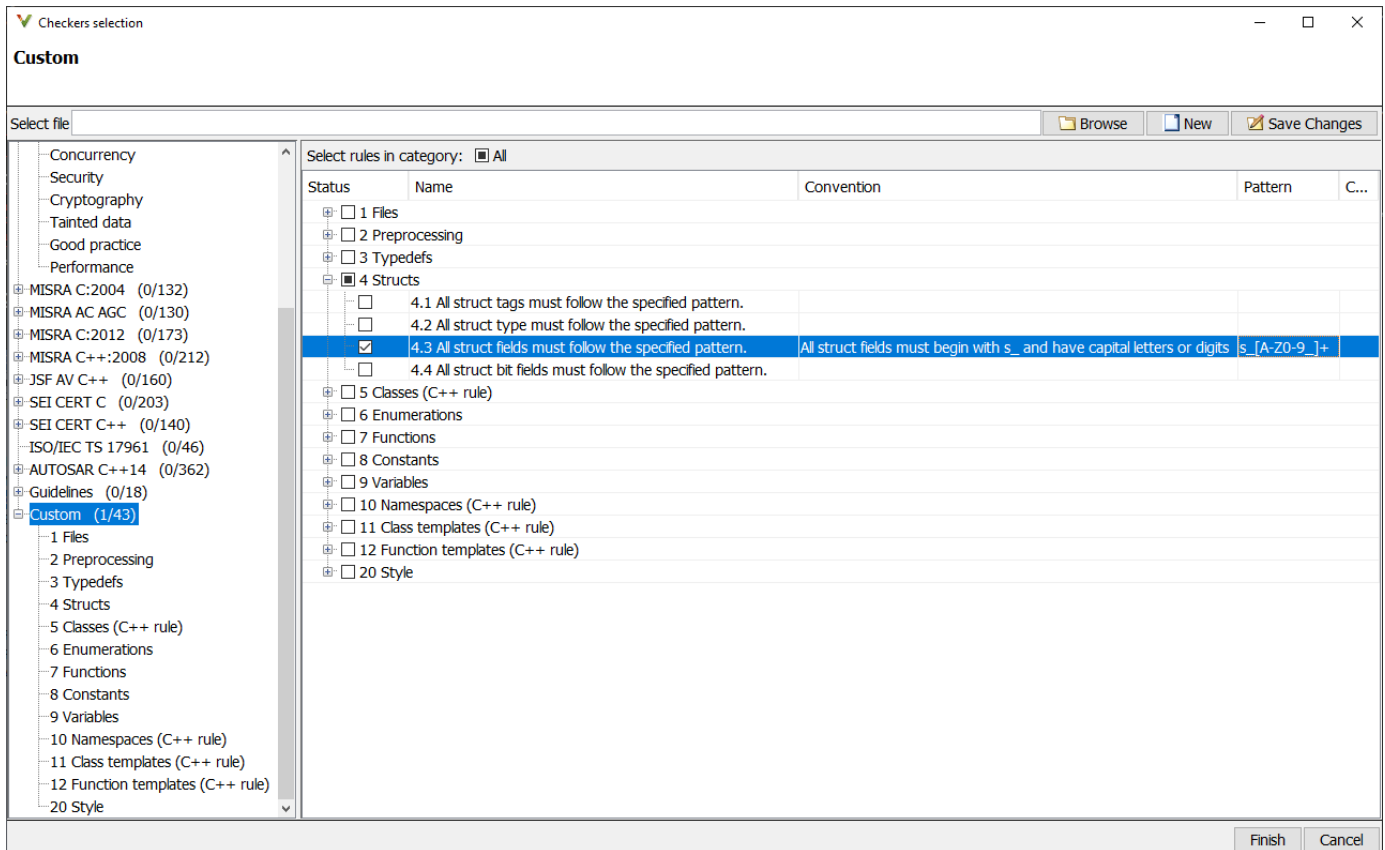
typedef struct {
    int a;
    int b;
} collection;

void main()
{
    collection myCollection= {0,0};
    printf("Initial values in the collection are %d and %d.",
        myCollection.a,myCollection.b);
}
```

Specify Naming Convention

Custom coding rule checkers compare the identifiers in your code to a naming convention that you specify. Polyspace raises a violation if the identifiers do not match the convention. Before you use the custom coding rules to enforce the naming convention, specify the naming convention by using a regular expression.

- 1 Open the Checkers Selection window. Depending on your workflow, you might open the window by using one of the Polyspace as You Code IDE plugins, the desktop user interface, or the command `polyspace-checkers-selection`.
- 2 In the Checkers Selection window, select the rule 4.3.
- 3 In the **Convention** field, enter the message that you want to display when the rule is violated. This message describes the naming convention that you want to specify. For instance, in the **Convention** field, enter `All struct fields must begin with s_ and have capital letters or digits`.
- 4 In the **Pattern** field, enter a regular expression corresponding to the naming convention that you want to specify. For instance, to represent `struct` field names that begin with `s_` and have capital letters or digits, specify `s_[A-Z0-9_]+`. Polyspace supports Perl regular expressions when defining patterns. See `Check custom rules (-custom-rules)`.



A custom rule is not activated if the **Pattern** field is empty.

- 5 The **Comment** field is optional. A comment does not appear in the Polyspace results list. Leave the **Comment** field blank.
- 6 Save your changes in an XML file and close the window. This XML file can be used to check the specified custom rule.

Alternatively, edit a preexisting checkers XML file to specify naming conventions. The Polyspace installation folder contains a template that you can copy and edit.

- 1 Locate the template `custom_rules.xml` in `polyspaceroot\help\toolbox\bugfinder\examples\coding_standards_XML`. Here, *polyspaceroot* is the root installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace Server\R2023a`. Make an editable copy of the file `custom_rules.xml`.
- 2 In the editable XML file, locate the node corresponding to rule 4.3. Set the state attribute to `on`. Add a subnode `Convention` and specify it as `All struct fields must begin with s_ and have capital letters or digits`. Then, add a subnode `Pattern` and specify it as `s_[A-Z0-9_]+`. For instance:

```
<check id="4.3" state="on">
  <convention>All struct fields must begin with s_
  and have capital letters or digits</convention>
  <pattern>s_[A-Z0-9_]+</pattern>
</check>
```

- 3 Save the XML file. You can use this XML file check the specified custom rule.

Check for Violations of Defined Custom Coding Rule

After specifying the naming convention, run a Polyspace analysis.

- If you are using the Polyspace desktop UI or one of the Polyspace as You Code plugins in an IDE, run a Polyspace analysis after saving your changes in the Checkers Selection window.
- If you are using the command line interface, provide the modified `custom_rules.xml` file as the argument for the option `Set checkers by file (-checkers-selection-file)` during analysis, along with the option `Check custom rules (-custom-rules)`. For instance, for custom rules checking by using Polyspace Bug Finder Server, enter:

```
polyspace-bug-finder-server -sources printInitialValue.c -custom-rules from-file  
-checkers-selection-file custom_rules.xml
```

The Polyspace analysis reports two violations of custom rule 4.3 on the two fields `collection.a` and `collection.b`.

To resolve the defects, change the name of the fields so that they comply with the naming convention. For instance, rename the fields as `s_A` and `s_B` respectively. After renaming the fields, run another Polyspace analysis to verify that the naming convention is no longer violated.

See Also

`Check custom rules (-custom-rules) | Set checkers by file (-checkers-selection-file)`

More About

- “Setting Checkers in Polyspace as You Code”
- “Run Polyspace Bug Finder on Desktop”
- Perl Regular Expression

Compute Code Complexity Metrics Using Polyspace

This topic describes how to compute all code complexity metrics and then review the ones that exceed specified thresholds. To see metric values only if they exceed thresholds, check for guidelines on software complexity. See “Reduce Software Complexity by Using Polyspace Checkers” on page 16-11.

Code complexity metrics are a set of numbers that quantify the complexity of your C/C++ program. For instance:

- A function with a high cyclomatic complexity contains too many branches.
- A function with a high number of return statements has too many exit points.

Complex programs are difficult to debug, analyze, test and maintain. To avoid too much complexity, impose limits on the complexity metrics during coding.

Polyspace does not compute code complexity metrics by default. To compute them during analysis, use the option `Calculate code metrics (-code-metrics)`.

After analysis, the software displays project, file and function metrics on the **Results List** pane. You can compare the computed metric values against predefined limits. If a metric value exceeds limits, you can redesign your code to lower the metric value. For instance, if the number of called functions is high and several of those functions are always called together, you can write one function that fuses the bodies of those functions. Call that one function instead of the group of functions that are called together.

Impose Limits on Metrics (Desktop Products Only)

In the user interface of the Polyspace desktop products, open some results with metrics computations. Then impose limits on the metric values and update results on the **Results List** pane to show only metric values that exceed the limits.

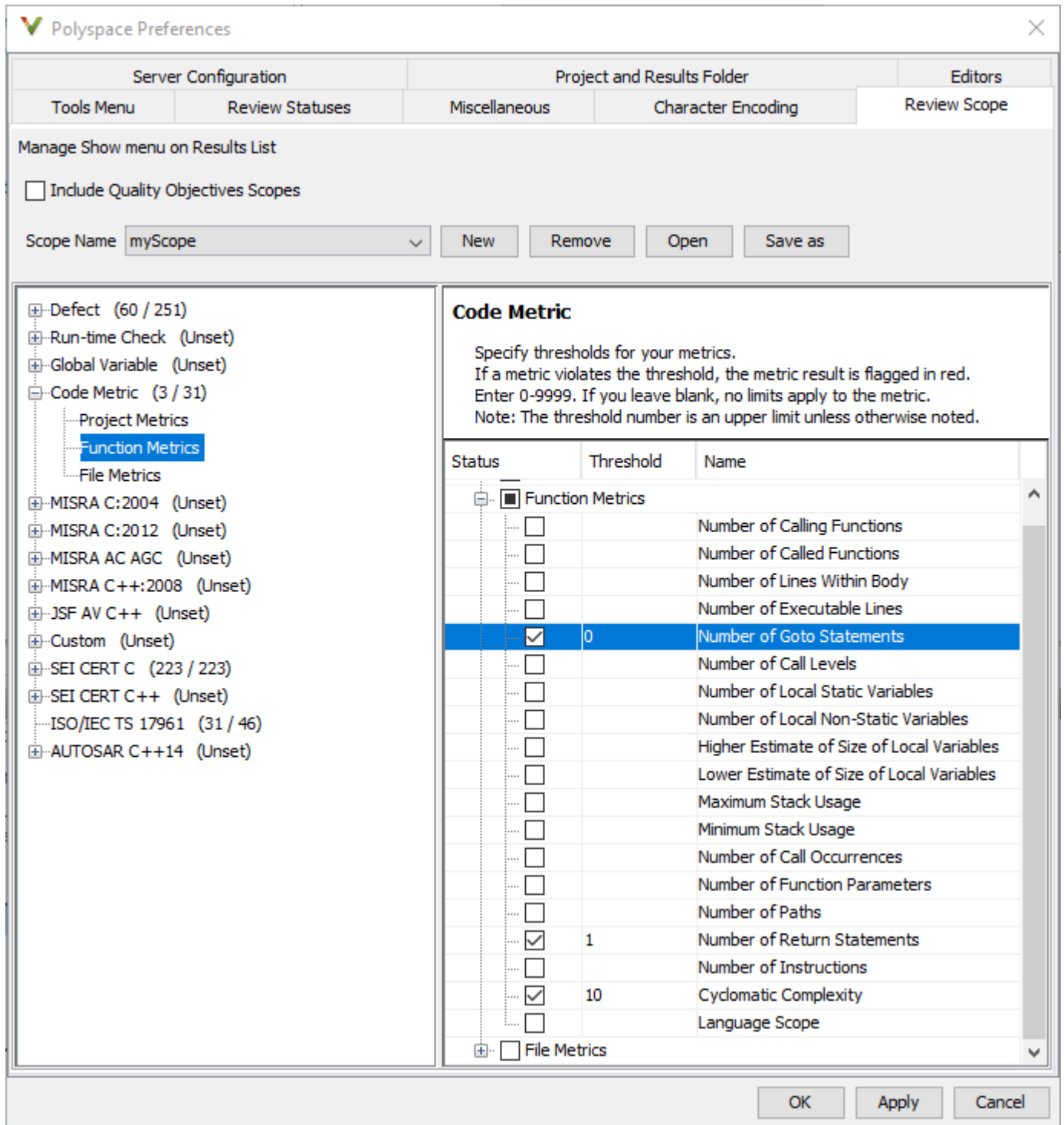
- 1 Select **Tools > Preferences**.
- 2 On the **Review Scope** tab, do one of the following:
 - To use a predefined limit, select **Include Quality Objectives Scopes**.

The **Scope Name** list shows the additional option **HIS**. The option **HIS** displays the **HIS** code metrics on page 16-54 only. Select the option to see the limit values.

- To define your own limits, select **New**. Save your limits file.

On the left pane, select **Code Metric**. On the right, select a metric and specify a limit value for the metric. Other than **Comment Density**, limit values are upper limits.

To select all metrics in a category such as **Function Metrics**, select the box next to the category name. For more information on the metrics categories, see “Code Metrics”. If only a some metrics in a category are selected, the check box next to the category name displays a symbol.



3 Select **Apply** or **OK**.

The drop-down list in the left of the **Results List** pane toolbar displays additional options.

- If you use predefined limits, the option HIS appears. This option displays code metrics only.

- If you define your own limits, the option corresponding to your limits file name appears.
- 4 Select the option corresponding to the limits that you want. Only metric values that violate your limits remain on the **Results List** pane.

These metrics are shown along with a red exclamation mark. For instance, the predefined scope, HIS, requires that every function should have only one return statement. If you select the scope HIS, you see the metric **Number of return statements** only if the number exceeds one.

Family	ID	Check: (1)	Information
! *	538	Number of Return Statements	Value: 2
! ★ *	1003	Number of Return Statements	Value: 2
! ★ *	1005	Number of Return Statements	Value: 2
! ★ *	1161	Number of Return Statements	Value: 2
! ★ *	1498	Number of Return Statements	Value: 3
! ★ *	2093	Number of Return Statements	Value: 3

- 5 Review each violation and decide how to rework your code to avoid the violation.

Note To enforce coding standards across your organization, share your limits file that you saved in XML format.

People in your organization can use the **Open** button on the **Review Scope** tab and navigate to the location of the XML file.

Impose Limits on Metrics (Server and Access products)

In the Polyspace Access web interface, limits on code complexity metrics are predefined. In the **Dashboard** perspective, if you select **Code Metric**, a **Code Metrics** window shows the metric values and limits.

To find the limits used, see “HIS Code Complexity Metrics” on page 16-54.

See also “Code Metrics Dashboard in Polyspace Access Web Interface” on page 25-11.

See Also

Calculate code metrics (-code-metrics)

More About

- “Code Metrics”
- “HIS Code Complexity Metrics” on page 16-54

HIS Code Complexity Metrics

The following list shows the Hersteller Initiative Software (HIS) standard metrics that Polyspace evaluates. These metrics and the recommended limits for their values are part of a standard defined by a major group of Original Equipment Manufacturers or OEMs. For more information on how to focus your review to this subset of code metrics, see “Compute Code Complexity Metrics Using Polyspace” on page 16-51.

Project

Polyspace evaluates the following HIS metrics at the project level.

Metric	Recommended Upper Limit
Number of direct recursions (AP_CG_DIRECT_CYCLE)	0
Number of recursions (AP_CG_CYCLE)	0

File

Polyspace evaluates the HIS metric, comment density, at the file level. The recommended lower limit is 20.

Function

Polyspace evaluates the following HIS metrics at the function level.

Metric	Recommended Upper Limit
Cyclomatic complexity (VG)	10
Language scope (VOCF)	4
Number of call levels (LEVEL)	4
Number of calling functions (CALLING)	5
Number of called functions (CALLS)	7
Number of function parameters (PARAM)	5
Number of goto statements (GOTO)	0
Number of instructions (STMT)	50
Number of paths (PATH)	80
Number of return statements (RETURN)	1

See Also

More About

- “Compute Code Complexity Metrics Using Polyspace” on page 16-51
- “Code Metrics”

Migrate Code Prover Workflows for Checking Coding Standards and Code Metrics to Bug Finder

In previous releases, Polyspace Code Prover supported checking of external coding standards and computation of code complexity metrics. For instance, you might be using Code Prover to:

- Check compliance with external coding standards such as MISRA C:2012 or MISRA C++:2008.
- Check compliance with naming conventions.
- Check compliance with code complexity standards.
- Calculate code metrics.

Support for the preceding capabilities will be removed from Code Prover in a future release. Starting in R2021b, Polyspace Bug Finder is the recommended tool for performing these tasks. Modify your workflows to migrate from using Code Prover to using Bug Finder.

Changes in Workflow

To migrate from Code Prover to Bug Finder, your workflow might need some changes.

Check for Coding Rule Violations and Compute Code Metrics

Previously, to perform this task, you configured the appropriate options in the **Configuration** pane, and then clicked **Run Code Prover**. To migrate to using Bug Finder:

- In the Polyspace user interface, configure the same options in the **Configuration** pane, and then click **Run Bug Finder**.
- At the command line, replace `polyspace-code-prover` by `polyspace-bug-finder`. If you do not want to enable the Bug Finder defects, specify `-checkers` with the value `none`. For instance, replace this command:

```
polyspace-code-prover -sources file_name -misra3 all -code-metrics
```

with this command:

```
polyspace-bug-finder -sources file_name -misra3 all -code-metrics -checkers none
```

Polyspace Bug Finder checks some coding rules differently compared to Code Prover. After migrating to Bug Finder, you might see some small difference in the number and location of coding rule violations.

Compute Code Metrics, Check for Run-Time Errors and Coding Rule Violations

Previously, to perform these tasks, you configured the appropriate options in the **Configuration** pane, and then clicked **Run Code Prover**. To migrate to using Bug Finder:

- In the Polyspace user interface, configure the same options in the **Configuration** pane. Then, obtain these results by performing two separate Polyspace analyses. Run a Bug Finder analysis to check for coding rule violations and to compute code metrics. Run a separate Code Prover verification to check for run-time errors.
- At the command line, run separate Bug Finder and Code Prover analyses by using the commands `polyspace-bug-finder` and `polyspace-code-prover` with appropriate analysis options. For instance, replace this command:

```
polyspace-code-prover -sources file_name -misra3 all -code-metrics
```

with this command:

```
polyspace-bug-finder -sources file_name -code-metrics -misra3 all -checkers none
polyspace-code-prover -sources file_name
```

Compute Code Metrics, Check for Run-Time Errors and Coding Rule Violations in Generated Code

Previously, to perform these tasks, you configured your Polyspace analysis, and then started a Code Prover verification. To migrate to using Bug Finder:

- On the Simulink toolstrip, use the same configurations that you used before. Then, run separate Bug Finder and Code Prover analyses.
- In the MATLAB Command Window, use separate sets of `polyspace.ModelLinkOptions` and `polyspace.Project` objects to perform separate Bug Finder and Code Prover analyses.

Sample MATLAB Code

```
% Make directory for code generation
[TEMPDIR, CGDIR] = rtwdemodir();
% Specify model name
modelName = 'rtwdemo_roll';
% Load the model
load_system(modelName);

% Set parameters for Embedded Coder target
set_param(modelName, 'SystemTargetFile', 'ert.tlc');
set_param(modelName, 'Solver', 'FixedStepDiscrete');
set_param(modelName, 'SupportContinuousTime', 'on');
set_param(modelName, 'LaunchReport', 'off');
set_param(modelName, 'InitFltsAndDblsToZero', 'on');

% Generate code
slbuild(modelName);

% Create Bug Finder project configuration
psprjCfgBF = polyspace.ModelLinkOptions(modelName);
% Enable coding rules, such as MISRA C:2012
psprjCfgBF.CodingRulesCodeMetrics.EnableMisraC3 = true;
psprjCfgBF.CodingRulesCodeMetrics.MisraC3Subset = 'all';
% Deactivate Bug Finder defects
psprjCfgBF.BugFinderAnalysis.EnableChecker = false;

% Specify results folder for bugfinder analysis
psprjCfgBF.ResultsDir = 'BF_newResfolder';

% Associate the project configurations with a Polyspace project
bfProj = polyspace.Project;
bfProj.Configuration = psprjCfgBF;

% Set verification mode as Bug Finder
bfStatus = bfProj.run('bugfinder');
% obtain BF results in a table
BF_results = bfProj.Results.getResults('readable');
```

```

% Create a new project for Code Prover verification
cpProj = polyspace.Project;

% Create a new configuration object for Code Prover verification
psprjCfgCP = polyspace.ModelLinkOptions(modelName);
psprjCfgCP.CodingRulesCodeMetrics.EnableMisraC3 = false;

% Specify results folder for Code Prover analysis
psprjCfgCP.ResultsDir = 'CP_newResfolder';

% Associate the project configurations with a Polyspace project
cpProj.Configuration = psprjCfgCP;

% Set verification mode as Code Prover
cpStatus = cpProj.run('codeprover');
% obtain CP results in a table
CP_results = cpProj.Results.getResults('readable');

```

Produce a Polyspace Report Containing Run-Time Errors, Coding Rule Violations, and Code Metrics

Previously, you configured a Code Prover verification to produce a single report containing run-time errors, code metrics, coding rule violations, and other results. To migrate to using Bug Finder, configure the same options and run separate Bug Finder and Code Prover analyses. See “Compute Code Metrics, Check for Run-Time Errors and Coding Rule Violations” on page 16-55.

The Bug Finder and Code Prover results are summarized in separated reports.

To produce a combined report containing Bug Finder and Code Prover results, use `polyspace-report-generator`. For instance, if your Bug Finder and Code Prover results are saved in the folders `BF_results` and `CP_results`, use this command at the command prompt:

```

polyspace-report-generator ^
-template %template_path% ^
-results-dir "CP_Results","BF_Results"

```

Sample Batch Script

```

@echo off
Rem Specify the path for source
set source=^
"C:\Program Files\Polyspace\R2021b\polyspace\examples\cxx\^
Bug_Finder_Example\sources\numerical.c"
Rem Using Developer.rpt as template
set template_path=^
"C:\Program Files\Polyspace\R2021b\toolbox\polyspace\psrptgen\^
templates\Developer.rpt"
Rem making results directory for Bug Finder and Code Prover Run
mkdir bfResults;
mkdir cpResults;
Rem Start Bug Finder analysis
polyspace-bug-finder -sources %source%^
-results-dir "%CD%\bfResults"^
-code-metrics ^
-misra3 all-rules ^
-lang c ^

```

```

-checkers none
Rem Start Code Prover analysis
polyspace-code-prover -sources %source%^
-results-dir "%CD%\cpResults"^
-lang c ^
-main-generator
Rem Start Report generation
polyspace-report-generator ^
-template %template_path% ^
-results-dir "%CD%\cpResults", "%CD%\bfResults"

```

Check for Protected and Unprotected Shared Global Variables

Previously, to perform this task, you specified the entry point functions and temporally exclusive functions in your code, and then computed code metrics by using Code Prover. The recommended tool for performing this task is to use the global variable checks in Code Prover instead.

- In the Polyspace user interface, configure the same options you did before, and then click **Run Code Prover**. You do not need to check **Calculate Code Metrics**.
- At the command line, run a Code Prover verification by using the same analysis options that you used before. Omit `-code-metrics`.

After the verification completes, in the Results List, the protected shared global variables are flagged by green checks, and the potentially unprotected shared global variables are flagged by orange checks.

Family	Information	File	Function	Status
Run-time Check		4 6		
Orange Check		4		
Green Check		6		
Global Variable		1 1		
Shared		1 1		
Potentially unprotected variable		1		
? x	Variable: shared_var	shared__global.c	_init_globals()	Unreviewed
Protected variable		1		
✓ x	Variable: Count	shared__global.c	_init_globals()	Unreviewed

See “Global Variables” (Polyspace Code Prover).

Calculate Stack Usage

Previously, to calculate stack usage, you configured your Code Prover analysis in the Polyspace UI to compute code metrics or specified the option `-code-metrics`. The recommended tool for performing this task is to use the analysis option `Calculate stack usage (-stack-usage)`.

- In the Polyspace user interface, check **Calculate stack usage** in the **Check Behavior** pane, and then click **Run Code Prover**. You do not need to check **Calculate Code Metrics**.
- At the command line, run a Code Prover verification by using the same analysis options that you used before. Use `-stack-usage` instead of `-code-metrics`.

After the verification completes, in the Results List, the stack usage metrics are listed.

See Also

More About

- “Justify Coding Rule Violations Using Code Prover Checks” (Polyspace Code Prover)
- “Check for and Review Coding Standard Violations” on page 16-2
- “AUTOSAR C++14 Rules”
- “CERT C Rules and Recommendations”
- “CERT C++ Rules”
- “Polyspace Support for Coding Standards” on page 17-2
- “Justify Coding Rule Violations Using Code Prover Checks” on page 31-7
- “Coding Standards & Code Metrics”

Polyspace Coverage of Coding Standards

- “Polyspace Support for Coding Standards” on page 17-2
- “MISRA C:2004 and MISRA AC AGC Coding Rules” on page 17-9
- “Required or Mandatory MISRA C:2012 Rules Supported by Polyspace Bug Finder” on page 17-43
- “Decidable MISRA C:2012 Rules Supported by Polyspace Bug Finder” on page 17-54
- “Undecidable MISRA C:2012 Rules and Directives Supported by Polyspace Bug Finder” on page 17-64
- “Polyspace Support for MISRA C: 2012 Amendments” on page 17-69
- “Essential Types in MISRA C:2012 Rules 10.x” on page 17-72
- “Unsupported MISRA C:2012 Guidelines” on page 17-74
- “Required and Statically Enforceable CERT C Rules Supported by Polyspace Bug Finder” on page 17-75
- “Required MISRA C++:2008 Coding Rules Supported by Polyspace Bug Finder” on page 17-83
- “JSF AV C++ Coding Rules” on page 17-100
- “Required AUTOSAR C++14 Coding Rules Supported by Polyspace Bug Finder” on page 17-125
- “Statically Enforceable AUTOSAR C++14 Rules Supported by Polyspace Bug Finder” on page 17-156

Polyspace Support for Coding Standards

Polyspace Bug Finder and Polyspace as You Code support various coding standards. Check the compliance of your code with these standards by analyzing your individual translation units in the IDE, and then analyzing your entire source code during integration. Polyspace as You Code supports a subset of rules that Bug Finder supports. See “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78.

Summary of Polyspace Support

Standard	Release	Statically Enforceable Rules	Required Mandatory Rules or
“AUTOSAR C++14” on page 17-2	10-31-2018	349 on page 17-156 out of 349 rules in the standard	337 on page 17-125 out of 362 rules in the standard
“MISRA C++:2008” on page 17-4	June 2008	^a	195 on page 17-83 out of 198 rules in the standard
“MISRA C:2012” on page 17-4	<ul style="list-style-type: none"> • March 2013 • April 2016 (Amendment 1) • June 2017 (TC1) • January 2018 (Amendment 2) 	122 on page 17-54 out of 122 rules in the standard	126 on page 17-43 out of 126 rules in the standard
“CERT C” on page 17-7	2016	120 on page 17-75 out of 120 rules in the standard	120 on page 17-75 out of 120 rules in the standard

^a MISRA C++:2008 standard does not categorize rules based on their static enforceability

Coding standards categorize the rules based on their obligation level or their static enforceability. Polyspace supports rules that are considered nonenforceable or partially enforceable by the standards. Enforcing these rules require a manual review process, which can be assisted by the Polyspace results.

AUTOSAR C++14

The AUTOSAR C++14 standard categorizes the rules based on their obligation level and enforcement by static analysis.

Obligation Level

Category	Rules Implemented in Bug Finder	Rules in the Standard
Required: The code must follow these rules.	337	362
Advisory: The code is advised to follow these rules to a reasonable practical extent.	33	35
	Total: 370	

Enforcement by Static Analysis Tool

Category	Rules Implemented in Bug Finder	Rules in the Standard
Automated: Static analysis tools can detect all violation of these rules.	327	327 ^a
Partially automated: Static analysis tools cannot detect all possible violations of these rules. You need manual code review or other tools to completely enforce these rules. Polyspace shows the subset of all possible issues. For details about which issues Polyspace detects for a particular rule, see the Polyspace Implementation section in the reference page of the rule.	22	22
Nonautomated: Static analysis tools cannot detect all possible violations of these rules. You need manual code review or other tools to completely enforce these rules. Polyspace shows the subset of all possible issues. For details about which issues Polyspace detects for a particular rule, see the Polyspace Implementation section in the reference page of the rule.	21	46

^a The AUTOSAR C++14 standard contains 329 **Automated** rules. The rules A0-4-3 and A1-4-3 are not enforceable by a static analysis tool. These rules might be enforced by a compiler.

The **Automated** and **Partially automated** rules are statically enforceable. In total, Polyspace supports 349 statically enforceable rules and 337 required rules. See:

- “Statically Enforceable AUTOSAR C++14 Rules Supported by Polyspace Bug Finder” on page 17-156.
- “Required AUTOSAR C++14 Coding Rules Supported by Polyspace Bug Finder” on page 17-125.

MISRA C++:2008

The MISRA C++:2008 standard categorizes the rules based on their obligation level.

Category	Rules Implemented in Bug Finder	Rules in the Standard
Required: The code must follow these rules.	195	198
Advisory: The code is advised to follow these rules to a reasonable practical extent.	18	18
Document: These rules are associated with different features including #pragmas, floating-point arithmetic, or bit fields. Whenever these features are used, the code must follow the associated rule.	1	12
	Total: 214	

In total Polyspace supports 193 out of 198 required MISRA C++:2008 rules. See “Required MISRA C++:2008 Coding Rules Supported by Polyspace Bug Finder” on page 17-83.

MISRA C:2012

The MISRA C:2012 standard classifies the guidelines as either a rule or a directive. Polyspace supports the original MISRA C:2012 standard as well as the technical corrigendum 1, amendments 1, and 2. See “Polyspace Support for MISRA C: 2012 Amendments” on page 17-69.

MISRA C:2012 Rules

A rule is a guideline that can be described completely. Compliance with a rule can be checked statically with some limitation. The rules are further categorized based on different properties.

Obligation Level

Category	Rules Implemented in Bug Finder	Rules in the Standard
Mandatory: These are guidelines that compliant C code must follow. The standard does not permit deviations from these guidelines.	16	16
Required: These are guidelines that compliant C code must follow. The standard permits only the deviations that you formally record and authorize.	110	110
Advisory: These are recommended guidelines. The standard permits deviation from these guidelines without any formal record. It is a best practice to follow these guidelines to a reasonably practical degree and record the deviations.	32	32
	Total: 158	

Compliant C code must follow the **Mandatory** and **Required** rules. Polyspace supports all such rules.

Static Enforceability

Category	Rules Implemented in Bug Finder	Rules in the Standard
Decidable: A rule is decidable if a static analysis tool can check compliance with the rule in every possible case.	122	122
Undecidable: A rule is undecidable if a static analysis tool can check compliance to it only in certain cases. Polyspace shows the subset of all possible issues. For details about which issues Polyspace detects for a particular rule, see the Polyspace Implementation section in the reference page of the rule.	36	36

Analysis Scope

Category	Rules Implemented in Bug Finder	Rules in the Standard
Single Translation Unit: You can find all violations of these rules by checking each translation unit of a project individually.	109	109
System: You can find all violations of these rules only by analyzing the entire project or system.	49	49

Polyspace supports 122 out of 122 decidable MISRA C:2012 rules. See “Decidable MISRA C:2012 Rules Supported by Polyspace Bug Finder” on page 17-54.

MISRA C:2012 Directives

Directives are guidelines that cannot be completely described. Checking compliance with these directives requires more information in addition to the code. Static analysis might assist in checking compliance with directives. The directives are categorized based on obligation level.

Obligation Level

Category	Directives Implemented in Bug Finder	Directives in the Standard
Required: These are guidelines that compliant C code must follow. The standard permits only the deviations that you formally record and authorize.	9	10
Advisory: These are recommended guidelines. The standard permits deviation from these guidelines without any formal record. It is a best practice to follow these guidelines to a reasonably practical degree and record the deviations.	6	7

Static Enforceability

Category	Directive Implemented in Bug Finder	Directive in the Standard
Decidable: A directive is decidable if a static analysis tool can check compliance with the directive in every possible case.	0	0
Undecidable: A directive is undecidable if a static analysis tool can check compliance to it only in certain cases. Polyspace shows the subset of all possible issues. For details about which issues Polyspace detects for a particular directive, see the Polyspace Implementation section in the reference page of the directive.	15	17

Polyspace supports 36 out of 36 undecidable rules as well as 15 undecidable directives. See “Undecidable MISRA C:2012 Rules and Directives Supported by Polyspace Bug Finder” on page 17-64.

CERT C

Polyspace supports all statically enforceable rules in the CERT C standard. The standard categorizes the guidelines into rules and recommendations. Polyspace does not support rules that are being removed or under construction.

Category	Checks Implemented in Bug Finder	Checks in the Standard
Rule: These guidelines are required. Violation of these guidelines might compromise the safety, security, or reliability of a system. Static analysis tools can enforce compliance with these guidelines.	120	120
Recommendation: These guidelines are meant to improve the readability, safety, and security of a system. Static analysis can only detect a subset of violations of these guidelines. Polyspace shows the subset of all possible issues. For details about which issues Polyspace detects, see the reference page of these rules.	83	183

Other

Polyspace also supports these coding rule standards.

Standard	Rules Implemented in Bug Finder
MISRA C:2004	132 out of 142 rules in the standard
MISRA AC AGC	130 out of 142 in the standard
ISO/IEC TS 17961	46 out of 46 in the standard
JSF AV C++	160 out of 234 in the standard
CERT C++	153 out of 163 in the standard
CWE	102 (version 4.9), including: <ul style="list-style-type: none"> • 72 out of 82 C specific rules (CWE-658). • 75 out of 86 C++ specific rules (CWE-659).

See Also

Check AUTOSAR C++ 14 (-autosar-cpp14) | Check MISRA C++:2008 (-misra-cpp) | Check SEI CERT-C++ (-cert-cpp) | Check MISRA C:2012 (-misra3) | Check MISRA C:2004 (-misra2) | Check SEI CERT-C (-cert-c) | Check CWE (-cwe)

More About

- “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78
- “Polyspace Support for MISRA C: 2012 Amendments” on page 17-69
- “Decidable MISRA C:2012 Rules Supported by Polyspace Bug Finder” on page 17-54
- “Undecidable MISRA C:2012 Rules and Directives Supported by Polyspace Bug Finder” on page 17-64
- “Required AUTOSAR C++14 Coding Rules Supported by Polyspace Bug Finder” on page 17-125
- “Statically Enforceable AUTOSAR C++14 Rules Supported by Polyspace Bug Finder” on page 17-156
- “Required MISRA C++:2008 Coding Rules Supported by Polyspace Bug Finder” on page 17-83
- “Required and Statically Enforceable CERT C Rules Supported by Polyspace Bug Finder” on page 17-75

MISRA C:2004 and MISRA AC AGC Coding Rules

In this section...

“Supported MISRA C:2004 and MISRA AC AGC Rules” on page 17-9

“Troubleshooting” on page 17-9

“List of Supported Coding Rules” on page 17-9

“Unsupported MISRA C:2004 and MISRA AC AGC Rules” on page 17-41

Supported MISRA C:2004 and MISRA AC AGC Rules

The following tables list MISRA C:2004 coding rules that the Polyspace coding rules checker supports. Details regarding how the software checks individual rules and any limitations on the scope of checking are described in the “Polyspace Specification” column.

Note The Polyspace coding rules checker:

- Supports MISRA-C:2004 Technical Corrigendum 1 for rules 4.1, 5.1, 5.3, 6.1, 6.3, 7.1, 9.2, 10.5, 12.6, 13.5, and 15.0.
- Checks rules specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

The software reports most violations during the compile phase of an analysis. However, the software detects violations of rules 9.1 (Non-initialized variable), 12.11 (one of the overflow checks) using `-scalar-overflows-checks signed-and-unsigned`, 13.7 (dead code), 14.1 (dead code), 16.2 and 21.1 during code analysis, and reports these violations as run-time errors.

Note Some violations of rules 13.7 and 14.1 are reported during the compile phase of analysis.

Troubleshooting

If you expect a rule violation but do not see it, check out “Diagnose Why Coding Standard Violations Do Not Appear as Expected” on page 32-49.

List of Supported Coding Rules

- “Environment” on page 17-11
- “Language Extensions” on page 17-12
- “Documentation” on page 17-15
- “Character Sets” on page 17-15
- “Identifiers” on page 17-15
- “Types” on page 17-16
- “Constants” on page 17-17

- “Declarations and Definitions” on page 17-17
- “Initialisation” on page 17-20
- “Arithmetic Type Conversion” on page 17-21
- “Pointer Type Conversion” on page 17-24
- “Expressions” on page 17-25
- “Control Statement Expressions” on page 17-27
- “Control Flow” on page 17-29
- “Switch Statements” on page 17-31
- “Functions” on page 17-32
- “Pointers and Arrays” on page 17-33
- “Structures and Unions” on page 17-34
- “Preprocessing Directives” on page 17-34
- “Standard Libraries” on page 17-37
- “Runtime Failures” on page 17-41

Environment

N.	MISRA Definition	Messages in report file	Polyspace Implementation
1.1	All code shall conform to ISO 9899:1990 "Programming languages - C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.	<p>The text <i>All code shall conform to ISO 9899:1990 Programming languages C, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996</i> precedes each of the following messages:</p> <ul style="list-style-type: none"> • ANSI C does not allow '#include_next' • ANSI C does not allow macros with variable arguments list • ANSI C does not allow '#assert' • ANSI C does not allow '#unassert' • ANSI C does not allow testing assertions • ANSI C does not allow '#ident' • ANSI C does not allow '#sccs' • text following '#else' violates ANSI standard. • text following '#endif' violates ANSI standard. • text following '#else' or '#endif' violates ANSI standard. 	All the supported extensions lead to a violation of this MISRA rule. Standard compilation error messages do not lead to a violation of this MISRA rule and remain unchanged.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
1.1 (cont.)		<p>The text <i>All code shall conform to ISO 9899:1990 Programming languages C, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996</i> precedes each of the following messages:</p> <ul style="list-style-type: none"> • ANSI C90 forbids 'long long int' type. • ANSI C90 forbids 'long double' type. • ANSI C90 forbids long long integer constants. • Keyword 'inline' should not be used. • Array of zero size should not be used. • Integer constant does not fit within unsigned long int. • Integer constant does not fit within long int. • Too many nesting levels of #includes: N_1. The limit is N_0. • Too many macro definitions: N_1. The limit is N_0. • Too many nesting levels for control flow: N_1. The limit is N_0. • Too many enumeration constants: N_1. The limit is N_0. 	

Language Extensions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
2.1	Assembly language shall be encapsulated and isolated.	Assembly language shall be encapsulated and isolated.	<p>No warnings if code is encapsulated in the following:</p> <ul style="list-style-type: none"> • asm functions or asm pragma • Macros

N.	MISRA Definition	Messages in report file	Polyspace Implementation
2.2	Source code shall only use /* */ style comments	C++ comments shall not be used.	C++ comments are handled as comments but lead to a violation of this MISRA rule Note: This rule cannot be annotated in the source code.
2.3	The character sequence /* shall not be used within a comment	The character sequence /* shall not appear within a comment.	This rule violation is also raised when the character sequence /* inside a C++ comment. Note: This rule cannot be annotated in the source code.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
2.4	Sections of code should not be "commented out"	Sections of code should not be "commented out"	<p>The checker uses internal heuristics to detect commented out code. For instance, characters such as #, ;, { or } indicate comments that might potentially contain code. These comments are then evaluated against other metrics to determine the likelihood of code masquerading as comment. For instance, several successive words without a symbol in between reduces this likelihood.</p> <p>The checker does not flag the following comments even if they contain code:</p> <ul style="list-style-type: none"> • Doxygen comments beginning with /** or /*!. • Comments that repeat the same symbol several times, for instance, the symbol = here: <pre data-bbox="1109 1081 1299 1165"> /** ===== * A comment * =====*/ </pre> • Comments on the first line of a file. • Comments that mix the C style (/* */) and C++ style (//). <p>The checker considers that these comments are meant for documentation purposes or entered deliberately with some forethought.</p>

Documentation

Rule	MISRA Definition	Messages in report file	Polyspace Implementation
3.4	All uses of the <i>#pragma</i> directive shall be documented and explained.	All uses of the <i>#pragma</i> directive shall be documented and explained.	To check this rule, you must list the pragmas that are allowed in source files by using the option <code>Allowed pragmas (-allowed-pragmas)</code> . If Polyspace finds a pragma not in the allowed pragma list, a violation is raised.

Character Sets

N.	MISRA Definition	Messages in report file	Polyspace Implementation
4.1	Only those escape sequences which are defined in the ISO C standard shall be used.	<code>\<character></code> is not an ISO C escape sequence Only those escape sequences which are defined in the ISO C standard shall be used.	
4.2	Trigraphs shall not be used.	Trigraphs shall not be used.	Trigraphs are handled and converted to the equivalent character but lead to a violation of the MISRA rule

Identifiers

N.	MISRA Definition	Messages in report file	Polyspace Implementation
5.1	Identifiers (internal and external) shall not rely on the significance of more than 31 characters	Identifier 'XX' should not rely on the significance of more than 31 characters.	All identifiers (global, static and local) are checked. For easier review, the rule checker shows all identifiers that have the same first 31 characters as one rule violation. You can see all instances of conflicting identifier names in the event history of that rule violation. <i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Analysis" on page 11-78.</i>
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	<ul style="list-style-type: none"> Local declaration of XX is hiding another identifier. Declaration of parameter XX is hiding another identifier. 	Assumes that rule 8.1 is not violated.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
5.3	A typedef name shall be a unique identifier	{typedef name}'%s' should not be reused. (already used as {typedef name} at %s:%d)	Warning when a typedef name is reused as another identifier name.
5.4	A tag name shall be a unique identifier	{tag name}'%s' should not be reused. (already used as {tag name} at %s:%d)	Warning when a tag name is reused as another identifier name <i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Analysis" on page 11-78.</i>
5.5	No object or function identifier with a static storage duration should be reused.	{static identifier/parameter name}'%s' should not be reused. (already used as {static identifier/parameter name} with static storage duration at %s:%d)	Warning when a static name is reused as another identifier name
5.6	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.	{member name}'%s' should not be reused. (already used as {member name} at %s:%d)	Warning when an idf in a namespace is reused in another namespace <i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Analysis" on page 11-78.</i>
5.7	No identifier name should be reused.	{identifier}'%s' should not be reused. (already used as {identifier} at %s:%d)	No violation reported when: <ul style="list-style-type: none"> • Different functions have parameters with the same name • Different functions have local variables with the same name • A function has a local variable that has the same name as a parameter of another function

Types

N.	MISRA Definition	Messages in report file	Polyspace Implementation
6.1	The plain char type shall be used only for the storage and use of character values	Only permissible operators on plain chars are '=', '==' or '!=' operators, explicit casts to integral types and '?' (for the 2nd and 3rd operands)	Warning when a plain char is used with an operator other than =, ==, !=, explicit casts to integral types, or as the second or third operands of the ? operator.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
6.2	Signed and unsigned char type shall be used only for the storage and use of numeric values.	<ul style="list-style-type: none"> Value of type plain char is implicitly converted to signed char. Value of type plain char is implicitly converted to unsigned char. Value of type signed char is implicitly converted to plain char. Value of type unsigned char is implicitly converted to plain char. 	Warning if value of type plain char is implicitly converted to value of type signed char or unsigned char.
6.3	<i>typedefs</i> that indicate size and signedness should be used in place of the basic types	<i>typedefs</i> that indicate size and signedness should be used in place of the basic types.	No warning is given in typedef definition.
6.4	Bit fields shall only be defined to be of type <i>unsigned int</i> or <i>signed int</i> .	Bit fields shall only be defined to be of type unsigned int or signed int.	
6.5	Bit fields of type <i>signed int</i> shall be at least 2 bits long.	Bit fields of type signed int shall be at least 2 bits long.	No warning on anonymous signed int bitfields of width 0 - Extended to all signed bitfields of size ≤ 1 (if Rule 6.4 is violated).

Constants

N.	MISRA Definition	Messages in report file	Polyspace Implementation
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.	<ul style="list-style-type: none"> Octal constants other than zero and octal escape sequences shall not be used. Octal constants (other than zero) should not be used. Octal escape sequences should not be used. 	

Declarations and Definitions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	<ul style="list-style-type: none"> Function XX has no complete prototype visible at call. Function XX has no prototype visible at definition. 	Prototype visible at call must be complete.
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated	Whenever an object or function is declared or defined, its type shall be explicitly stated.	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
8.3	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.	Definition of function 'XX' incompatible with its declaration.	Assumes that rule 8.1 is not violated. The rule is restricted to compatible types. Can be turned to Off
8.4	If objects or functions are declared more than once their types shall be compatible.	<ul style="list-style-type: none"> • If objects or functions are declared more than once their types shall be compatible. • Global declaration of 'XX' function has incompatible type with its definition. • Global declaration of 'XX' variable has incompatible type with its definition. 	Violations of this rule might be generated during the link phase. <i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Analysis" on page 11-78.</i>
8.5	There shall be no definitions of objects or functions in a header file	<ul style="list-style-type: none"> • Object 'XX' should not be defined in a header file. • Function 'XX' should not be defined in a header file. • Fragment of function should not be defined in a header file. 	Tentative definitions are considered as definitions. For objects with file scope, tentative definitions are declarations that: <ul style="list-style-type: none"> • Do not have initializers. • Do not have storage class specifiers, or have the <code>static</code> specifier
8.6	Functions shall always be declared at file scope.	Function 'XX' should be declared at file scope.	This rule maps to ISO/IEC TS 17961 ID addresscape.
8.7	Objects shall be defined at block scope if they are only accessed from within a single function	Object 'XX' should be declared at block scope.	Restricted to static objects.
8.8	An external object or function shall be declared in one file and only one file	Function/Object 'XX' has external declarations in multiple files.	Restricted to explicit extern declarations (tentative definitions are ignored). Polyspace considers that variables or functions declared <code>extern</code> in a non-header file violate this rule.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
8.9	An identifier with external linkage shall have exactly one external definition.	<ul style="list-style-type: none"> • Procedure/Global variable XX multiply defined. • Forbidden multiple tentative definitions for object XX • Global variable has multiple tentative definitions • Undefined global variable XX 	<p>The checker flags multiple definitions only if the definitions occur in different files.</p> <p>No warnings appear on predefined symbols.</p> <p><i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Analysis" on page 11-78.</i></p>
8.10	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required	Function/Variable XX should have internal linkage.	<p>Assumes that 8.1 is not violated. No warning if 0 uses.</p> <p>If your code does not contain a main function and you use options such as Variables to initialize (-main-generator-writes-variables) with value custom to explicitly specify a set of variables to initialize, the checker does not flag those variables. The checker assumes that in a real application, the file containing the main must initialize the variables in addition to any file that currently uses them. Therefore, the variables must be used in more than one translation unit.</p> <p><i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Analysis" on page 11-78.</i></p>
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage	static storage class specifier should be used on internal linkage symbol XX.	
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization	Size of array 'XX' should be explicitly stated.	

Initialisation

N.	MISRA Definition	Messages in report file	Polyspace Implementation
9.1	All automatic variables shall have been assigned a value before being used.		Polyspace reports a violation of this rule if your code contains these issues: <ul style="list-style-type: none"> • Non-initialized variable • Non-initialized pointer
9.2	Braces shall be used to indicate and match the structure in the nonzero initialisation of arrays and structures.	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.	
9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	

Arithmetic Type Conversion

N.	MISRA Definition	Messages in report file	Polyspace Implementation
10.1	<p>The value of an expression of integer type shall not be implicitly converted to a different underlying type if:</p> <ul style="list-style-type: none"> • it is not a conversion to a wider integer type of the same signedness, or • the expression is complex, or • the expression is not constant and is a function argument, or • the expression is not constant and is a return expression 	<ul style="list-style-type: none"> • Implicit conversion of the expression of underlying type XX to the type XX that is not a wider integer type of the same signedness. • Implicit conversion of one of the binary operands whose underlying types are XX and XX • Implicit conversion of the binary right hand operand of underlying type XX to XX that is not an integer type. • Implicit conversion of the binary left hand operand of underlying type XX to XX that is not an integer type. • Implicit conversion of the binary right hand operand of underlying type XX to XX that is not a wider integer type of the same signedness or • Implicit conversion of the binary ? left hand operand of underlying type XX to XX, but it is a complex expression. • Implicit conversion of complex integer expression of underlying type XX to XX. • Implicit conversion of non-constant integer expression of underlying type XX in function return whose expected type is XX. • Implicit conversion of non-constant integer expression of underlying type XX as argument of function whose corresponding parameter type is XX. 	<p>ANSI C base types order (signed char, short, int, long) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1. The same interpretation is applied on the unsigned version of base types.</p> <p>An expression of bool or enum types has int as underlying type.</p> <p>Plain char may have signed or unsigned underlying type (depending on Polyspace target configuration or option setting).</p> <p>The underlying type of a simple expression of struct.bitfield is the base type used in the bitfield definition, the bitfield width is not token into account and it assumes that only signed unsigned int are used for bitfield (Rule 6.4).</p> <p>This rule violation is not produced on operations involving pointers.</p> <p>No violation reported when:</p> <ul style="list-style-type: none"> • The implicit conversion is a type widening, without change of signedness of integer • The expression is an argument expression or a return expression <p>No violation reported when the following are true:</p> <ul style="list-style-type: none"> • Implicit conversion applies to a constant expression and is a type widening, with a possible change of signedness of integer. • The conversion does not change the representation of

N.	MISRA Definition	Messages in report file	Polyspace Implementation
			<p>the constant value or the result of the operation.</p> <ul style="list-style-type: none"> The expression is an argument expression or a return expression or an operand expression of a non-bitwise operator. <p>Conversions of constants are not reported for these cases to avoid flagging too many violations. If the constant can be represented in both the original and converted type, the conversion is less of an issue.</p>
10.2	<p>The value of an expression of floating type shall not be implicitly converted to a different type if</p> <ul style="list-style-type: none"> it is not a conversion to a wider floating type, or the expression is complex, or the expression is a function argument, or the expression is a return expression 	<ul style="list-style-type: none"> Implicit conversion of the expression from XX to XX that is not a wider floating type. Implicit conversion of the binary ? right hand operand from XX to XX, but it is a complex expression. Implicit conversion of the binary ? right hand operand from XX to XX that is not a wider floating type or Implicit conversion of the binary ? left hand operand from XX to XX, but it is a complex expression. Implicit conversion of complex floating expression from XX to XX. Implicit conversion of floating expression of XX type in function return whose expected type is XX. Implicit conversion of floating expression of XX type as argument of function whose corresponding parameter type is XX. 	<p>ANSI C base types order (float, double) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1.</p> <p>No violation reported when:</p> <ul style="list-style-type: none"> The implicit conversion is a type widening The expression is an argument expression or a return expression.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression	Complex expression of underlying type XX may only be cast to narrower integer type of same signedness, however the destination type is XX.	<ul style="list-style-type: none"> • The rule checker raises a defect only if the result of a composite expression is cast to a different or wider essential type. For instance, in this example, a violation is shown in the first assignment to <code>i</code> but not the second. In the first assignment, a composite expression <code>i+1</code> is directly cast from a signed to an unsigned type. In the second assignment, the composite expression is first cast to the same type and then the result is cast to a different type. <pre>typedef int int32_T; typedef unsigned char uint8_T; int32_T i; i = (uint8_T)(i+1); /* Noncompliant */ i = (uint8_T) ((int32_T)(i+1)); /* Compliant */</pre> • ANSI C base types order (signed char, short, int, long) defines that T1 is narrower than T2 if T2 is on the right hand of T1 or T1 = T2. The same methodology is applied on the unsigned version of base types. • An expression of bool or enum types has int as underlying type. • Plain char may have signed or unsigned underlying type (depending on target configuration or option setting). • The underlying type of a simple expression of struct.bitfield is the base type used in the bitfield definition, the bitfield width

N.	MISRA Definition	Messages in report file	Polyspace Implementation
			is not taken into account and it assumes that only signed, unsigned int are used for bitfield (Rule 6.4).
10.4	The value of a complex expression of float type may only be cast to narrower floating type	Complex expression of XX type may only be cast to narrower floating type, however the destination type is XX.	ANSI C base types order (float, double) defines that T1 is narrower than T2 if T2 is on the right hand of T1 or T2 = T1.
10.5	If the bitwise operator ~ and << are applied to an operand of underlying type <i>unsigned char</i> or <i>unsigned short</i> , the result shall be immediately cast to the underlying type of the operand	Bitwise [<< ~] is applied to the operand of underlying type [unsigned char unsigned short], the result shall be immediately cast to the underlying type.	
10.6	The "U" suffix shall be applied to all constants of <i>unsigned</i> types	No explicit 'U' suffix on constants of an unsigned type.	Warning when the type determined from the value and the base (octal, decimal or hexadecimal) is unsigned and there is no suffix u or U. For example, when the size of the <code>int</code> and <code>long int</code> data types is 32 bits, the coding rule checker will report a violation of rule 10.6 for the following line: <code>int a = 2147483648;</code> There is a difference between decimal and hexadecimal constants when <code>int</code> and <code>long int</code> are not the same size.

Pointer Type Conversion

N.	MISRA Definition	Messages in report file	Polyspace Implementation
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type	Conversion shall not be performed between a pointer to a function and any type other than an integral type.	Casts and implicit conversions involving a function pointer. Casts or implicit conversions from NULL or (void*)0 do not give any warning.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.	There is also a warning on qualifier loss This rule maps to ISO/IEC TS 17961 ID <code>alignconv</code> .

N.	MISRA Definition	Messages in report file	Polyspace Implementation
11.3	A cast should not be performed between a pointer type and an integral type	A cast should not be performed between a pointer type and an integral type.	Exception on zero constant. Extended to all conversions This rule maps to ISO/IEC TS 17961 ID alignconv .
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.	A cast should not be performed between a pointer to object type and a different pointer to object type.	
11.5	A cast shall not be performed that removes any <i>const</i> or <i>volatile</i> qualification from the type addressed by a pointer	A cast shall not be performed that removes any <i>const</i> or <i>volatile</i> qualification from the type addressed by a pointer	Extended to all conversions

Expressions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
12.1	Limited dependence should be placed on C's operator precedence rules in expressions	Limited dependence should be placed on C's operator precedence rules in expressions	
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits.	<ul style="list-style-type: none"> The value of '<i>sym</i>' depends on the order of evaluation. The value of volatile '<i>sym</i>' depends on the order of evaluation because of multiple accesses. 	<p>Rule 12.2 check assumes that no assignment in expressions that yield a Boolean values (rule 13.1).</p> <p>The expression is a simple expression of symbols. <code>i = i++;</code> is a violation, but <code>tab[2] = tab[2]++;</code> is not a violation.</p>
12.3	The <code>sizeof</code> operator should not be used on expressions that contain side effects.	The <code>sizeof</code> operator should not be used on expressions that contain side effects.	No warning on volatile accesses
12.4	The right hand operand of a logical <code>&&</code> or <code> </code> operator shall not contain side effects.	The right hand operand of a logical <code>&&</code> or <code> </code> operator shall not contain side effects.	No warning on volatile accesses
12.5	The operands of a logical <code>&&</code> or <code> </code> shall be primary-expressions.	<ul style="list-style-type: none"> operand of logical <code>&&</code> is not a primary expression operand of logical <code> </code> is not a primary expression The operands of a logical <code>&&</code> or <code> </code> shall be primary-expressions. 	<p>During preprocessing, violations of this rule are detected on the expressions in <code>#if</code> directives.</p> <p>Allowed exception on associatively (<code>a && b && c</code>), (<code>a b c</code>).</p>

N.	MISRA Definition	Messages in report file	Polyspace Implementation
12.6	Operands of logical operators (&&, and !) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (&&, or !).	<ul style="list-style-type: none"> • Operand of '!' logical operator should be effectively Boolean. • Left operand of '%s' logical operator should be effectively Boolean. • Right operand of '%s' logical operator should be effectively Boolean. • %s operand of '%s' is effectively Boolean. Boolean should not be used as operands to operators other than '&&', ' ', '!', '=', '==', '!=', and '?:'. 	<p>The operand of a logical operator should be a Boolean data type. Although the C standard does not explicitly define the Boolean data type, the standard implicitly assumes the use of the Boolean data type.</p> <p>Some operators may return Boolean-like expressions, for example, <code>(var == 0)</code>.</p> <p>Consider the following code:</p> <pre>unsigned char flag; if (!flag)</pre> <p>The rule checker reports a violation of rule 12.6:</p> <pre>Operand of '!' logical operator should be effectively Boolean.</pre> <p>The operand <code>flag</code> is not a Boolean but an <code>unsigned char</code>.</p> <p>To be compliant with rule 12.6, the code must be rewritten either as</p> <pre>if (!(flag != 0)) or if (flag == 0)</pre> <p>The use of the option <code>-boolean-types</code> may increase or decrease the number of warnings generated.</p>
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed	<ul style="list-style-type: none"> • [<code>~/Left Shift/Right shift/&</code>] operator applied on an expression whose underlying type is signed. • Bitwise <code>~</code> on operand of signed underlying type <code>XX</code>. • Bitwise [<code><< >></code>] on left hand operand of signed underlying type <code>XX</code>. • Bitwise [<code>& ^</code>] on two operands of <code>s</code> 	<p>The underlying type for an integer is signed when:</p> <ul style="list-style-type: none"> • it does not have a <code>u</code> or <code>U</code> suffix • it is small enough to fit into a 64 bits signed number

N.	MISRA Definition	Messages in report file	Polyspace Implementation
12.8	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.	<ul style="list-style-type: none"> • shift amount is negative • shift amount is bigger than 64 • Bitwise [<< >>] count out of range [0 ..X] (width of the underlying type XX of the left hand operand - 1).. 	<p>The numbers that are manipulated in preprocessing directives are 64 bits wide so that valid shift range is between 0 and 63</p> <p>Check is also extended onto bitfields with the field width or the width of the base type when it is within a complex expression</p>
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	<ul style="list-style-type: none"> • Unary - on operand of unsigned underlying type XX. • Minus operator applied to an expression whose underlying type is unsigned 	<p>The underlying type for an integer is signed when:</p> <ul style="list-style-type: none"> • it does not have a u or U suffix • it is small enough to fit into a 64 bits signed number
12.10	The comma operator shall not be used.	The comma operator shall not be used.	
12.11	Evaluation of constant unsigned expression should not lead to wraparound.	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	
12.12	The underlying bit representations of floating-point values shall not be used.	The underlying bit representations of floating-point values shall not be used.	<p>Warning when:</p> <ul style="list-style-type: none"> • A float pointer is cast as a pointer to another data type. Casting a float pointer as a pointer to void does not generate a warning. • A float is packed with another data type. For example: <pre>union { float f; int i; } ...</pre>
12.13	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression	Warning when ++ or -- operators are not used alone.

Control Statement Expressions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
13.1	Assignment operators shall not be used in expressions that yield Boolean values.	Assignment operators shall not be used in expressions that yield Boolean values.	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	No warning is given on integer constants. Example: if (2) The use of the option <code>-boolean-types</code> may increase or decrease the number of warnings generated.
13.3	Floating-point expressions shall not be tested for equality or inequality.	Floating-point expressions shall not be tested for equality or inequality.	Warning on direct tests only.
13.4	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type	If <i>for</i> index is a variable symbol, checked that it is not a float.
13.5	The three expressions of a <i>for</i> statement shall be concerned only with loop control	<ul style="list-style-type: none"> • 1st expression should be an assignment. • Bad type for loop counter (XX). • 2nd expression should be a comparison. • 2nd expression should be a comparison with loop counter (XX). • 3rd expression should be an assignment of loop counter (XX). • 3rd expression: assigned variable should be the loop counter (XX). • The following kinds of <i>for</i> loops are allowed: <ul style="list-style-type: none"> (a) all three expressions shall be present; (b) the 2nd and 3rd expressions shall be present with prior initialization of the loop counter; (c) all three expressions shall be empty for a deliberate infinite loop. 	Checked if the <i>for</i> loop index (V) is a variable symbol; checked if V is the last assigned variable in the first expression (if present). Checked if, in first expression, if present, is assignment of V; checked if in 2nd expression, if present, must be a comparison of V; Checked if in 3rd expression, if present, must be an assignment of V.
13.6	Numeric variables being used within a <i>for</i> loop for iteration counting should not be modified in the body of the loop.	Numeric variables being used within a <i>for</i> loop for iteration counting should not be modified in the body of the loop.	Detect only direct assignments if the <i>for</i> loop index is known and if it is a variable symbol.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
13.7	Boolean operations whose results are invariant shall not be permitted	<ul style="list-style-type: none"> • Boolean operations whose results are invariant shall not be permitted. Expression is always true. • Boolean operations whose results are invariant shall not be permitted. Expression is always false. • Boolean operations whose results are invariant shall not be permitted. 	<p>During compilation, the checker covers comparisons with at least one constant operand. Some violations of this rule are reported through the <code>Dead code</code> and <code>Useless if</code> checkers.</p> <p>The rule violation appears when you check whether an enum variable value lies between its lower and upper bound. The violation appears even if you increment or decrement the variable outside its bounds, for instance, in this <code>for</code> loop condition:</p> <pre>enum ec {RED, BLUE, GREEN} col; for(col=RED; col<=GREEN; col++) {}</pre> <p>An enum variable can potentially wrap around when incremented outside its range and the loop condition can be always true. To avoid the rule violation, you can cast the enum to an integer before the comparison, for instance:</p> <pre>enum ec {RED, BLUE, GREEN} col; for(col=RED; (int)col<=GREEN; col++) {}</pre>

Control Flow

N.	MISRA Definition	Messages in report file	Polyspace Implementation
14.1	There shall be no unreachable code.	There shall be no unreachable code.	
14.2	All non-null statements shall either have at least one side effect however executed, or cause control flow to change	<p>All non-null statements shall either:</p> <ul style="list-style-type: none"> • have at least one side effect however executed, or • cause control flow to change 	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
14.3	Before preprocessing, a null statement shall occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.	A null statement shall appear on a line by itself	<p>We assume that a ';' is a null statement when it is the first character on a line (excluding comments). The rule is violated when:</p> <ul style="list-style-type: none"> • there are some comments before it on the same line. • there is a comment immediately after it • there is something else than a comment after the ';' on the same line.
14.4	The <i>goto</i> statement shall not be used.	The goto statement shall not be used.	
14.5	The <i>continue</i> statement shall not be used.	The continue statement shall not be used.	
14.6	For any iteration statement there shall be at most one <i>break</i> statement used for loop termination	For any iteration statement there shall be at most one break statement used for loop termination	
14.7	A function shall have a single point of exit at the end of the function	A function shall have a single point of exit at the end of the function	
14.8	The statement forming the body of a <i>switch</i> , <i>while</i> , <i>do while</i> or <i>for</i> statement shall be a compound statement	<ul style="list-style-type: none"> • The body of a do while statement shall be a compound statement. • The body of a for statement shall be a compound statement. • The body of a switch statement shall be a compound statement 	
14.9	An <i>if (expression)</i> construct shall be followed by a compound statement. The <i>else</i> keyword shall be followed by either a compound statement, or another <i>if</i> statement	<ul style="list-style-type: none"> • An if (expression) construct shall be followed by a compound statement. • The else keyword shall be followed by either a compound statement, or another if statement 	
14.10	All <i>if else if</i> constructs should contain a final <i>else</i> clause.	All if else if constructs should contain a final else clause.	

Switch Statements

N.	MISRA Definition	Messages in report file	Polyspace Implementation
15.0	The MISRA C switch syntax shall be used.	switch statements syntax normative restrictions.	<p>Warning on declarations or any statements before the first switch case.</p> <p>Warning on label or jump statements in the body of switch cases.</p> <p>On the following example, the rule is displayed in the log file at line 3:</p> <pre> 1 ... 2 switch(index) { 3 var = var + 1; // RULE 15.0 // violated 4case 1: ... </pre> <p>The code between switch statement and first case is checked as dead code by Polyspace. It follows ANSI standard behavior.</p> <p>This rule is not considered as a required rule in the MISRA C:2004 rules for generated code. In generated code, if you find a violation of rule 15.0 that does not simultaneously violate a later rule in this group, justify the violation with appropriate comments.</p>
15.1	A switch label shall only be used when the most closely-enclosing compound statement is the body of a <i>switch</i> statement	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	
15.2	An unconditional <i>break</i> statement shall terminate every non-empty switch clause	An unconditional break statement shall terminate every non-empty switch clause	Warning for each non-compliant case clause.
15.3	The final clause of a <i>switch</i> statement shall be the <i>default</i> clause	The final clause of a switch statement shall be the default clause	
15.4	A <i>switch</i> expression should not represent a value that is effectively Boolean	A switch expression should not represent a value that is effectively Boolean	The use of the option <code>-boolean-types</code> may increase the number of warnings generated.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
15.5	Every <i>switch</i> statement shall have at least one <i>case</i> clause	Every switch statement shall have at least one case clause	

Functions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
16.1	Functions shall not be defined with variable numbers of arguments.	Function XX should not be defined as varargs.	
16.2	Functions shall not call themselves, either directly or indirectly.	Function %s should not call itself.	The checker reports each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported. You can calculate the total number of recursion cycles using the code complexity metric Number of Recursions .
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.	Identifiers shall be given for all of the parameters in a function prototype declaration.	Assumes Rule 8.6 is not violated.
16.4	The identifiers used in the declaration and definition of a function shall be identical.	The identifiers used in the declaration and definition of a function shall be identical.	Assumes that rules 8.8 , 8.1 and 16.3 are not violated. All occurrences are detected.
16.5	Functions with no parameters shall be declared with parameter type <i>void</i> .	Functions with no parameters shall be declared with parameter type <i>void</i> .	Definitions are also checked.
16.6	The number of arguments passed to a function shall match the number of parameters.	<ul style="list-style-type: none"> • Too many arguments to XX. • Insufficient number of arguments to XX. 	Assumes that rule 8.1 is not violated. This rule maps to ISO/IEC TS 17961 ID argcomp .
16.7	A pointer parameter in a function prototype should be declared as <i>pointer</i> to <i>const</i> if the pointer is not used to modify the addressed object.	Pointer parameter in a function prototype should be declared as <i>pointer</i> to <i>const</i> if the pointer is not used to modify the addressed object.	Warning if a non- <i>const</i> pointer parameter is either not used to modify the addressed object or is passed to a call of a function that is declared with a <i>const</i> pointer parameter.
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Missing return value for non-void function XX.	Warning when a non-void function is not terminated with an unconditional return with an expression.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty.	Function identifier XX should be preceded by a & or followed by a parameter list.	
16.10	If a function returns error information, then that error information shall be tested.	If a function returns error information, then that error information shall be tested.	<p>The checker flags functions with non-void return if the return value is not used or not explicitly cast to a void type.</p> <p>The checker does not flag the functions memcopy, memset, memmove, strcpy, strncpy, strcat, strncat because these functions simply return a pointer to their first arguments.</p>

Pointers and Arrays

N.	MISRA Definition	Messages in report file	Polyspace Implementation
17.1	Pointer arithmetic shall only be applied to pointers that address an array or array element.	Pointer arithmetic shall only be applied to pointers that address an array or array element.	
17.2	Pointer subtraction shall only be applied to pointers that address elements of the same array	Pointer subtraction shall only be applied to pointers that address elements of the same array.	Polyspace reports a violation when you subtract that are null or that point to elements in different arrays.
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	Polyspace reports a violation when you compare pointers that are null or that point to elements in different arrays. The relational operators for the comparison are >, <, >=, and <=.
17.4	Array indexing shall be the only allowed form of pointer arithmetic.	Array indexing shall be the only allowed form of pointer arithmetic.	<p>Warning on:</p> <ul style="list-style-type: none"> Operations on pointers. (p+I, I+p, and p - I, where p is a pointer and I an integer). Array indexing on nonarray pointers.
17.5	A type should not contain more than 2 levels of pointer indirection	A type should not contain more than 2 levels of pointer indirection	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.	Pointer to a parameter is an illegal return value. Pointer to a local is an illegal return value.	Warning when assigning address to a global variable, returning a local variable address, or returning a parameter address. This rule maps to ISO/IEC TS 17961 ID accfree .

Structures and Unions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
18.1	All structure or union types shall be complete at the end of a translation unit.	All structure or union types shall be complete at the end of a translation unit.	Warning for all incomplete declarations of structs or unions.
18.2	An object shall not be assigned to an overlapping object.	<ul style="list-style-type: none"> An object shall not be assigned to an overlapping object. Destination and source of XX overlap, the behavior is undefined. 	
18.4	Unions shall not be used	Unions shall not be used.	

Preprocessing Directives

N.	MISRA Definition	Messages in report file	Polyspace Implementation
19.1	<code>#include</code> statements in a file shall only be preceded by other preprocessors directives or comments	<code>#include</code> statements in a file shall only be preceded by other preprocessors directives or comments	A message is displayed when a <code>#include</code> directive is preceded by other things than preprocessor directives, comments, spaces or "new lines".
19.2	Nonstandard characters should not occur in header file names in <code>#include</code> directives	<ul style="list-style-type: none"> A message is displayed on characters ', " or /* between < and > in <code>#include <filename></code> A message is displayed on characters ', or /* between " and " in <code>#include "filename"</code> 	
19.3	The <code>#include</code> directive shall be followed by either a <code><filename></code> or <code>"filename"</code> sequence.	<ul style="list-style-type: none"> <code>'#include'</code> expects <code>"FILENAME"</code> or <code><FILENAME></code> <code>'#include_next'</code> expects <code>"FILENAME"</code> or <code><FILENAME></code> 	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.	Macro '<name>' does not expand to a compliant construct.	<p>We assume that a macro definition does not violate this rule when it expands to:</p> <ul style="list-style-type: none"> • a braced construct (not necessarily an initializer) • a parenthesized construct (not necessarily an expression) • a number • a character constant • a string constant (can be the result of the concatenation of string field arguments and literal strings) • the following keywords: typedef, extern, static, auto, register, const, volatile, __asm__ and __inline__ • a do-while-zero construct
19.5	Macros shall not be #defined and #undef'd within a block.	<ul style="list-style-type: none"> • Macros shall not be #define'd within a block. • Macros shall not be #undef'd within a block. 	
19.6	#undef shall not be used.	#undef shall not be used.	
19.7	A function should be used in preference to a function like-macro.	A function should be used in preference to a function like-macro	Message on all function-like macro definitions.
19.8	A function-like macro shall not be invoked without all of its arguments	<ul style="list-style-type: none"> • arguments given to macro '<name>' • macro '<name>' used without args. • macro '<name>' used with just one arg. • macro '<name>' used with too many (<number>) args. 	
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	Macro argument shall not look like a preprocessing directive.	This rule is detected as violated when the '#' character appears in a macro argument (outside a string or character constant)

N.	MISRA Definition	Messages in report file	Polyspace Implementation
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.	Parameter instance shall be enclosed in parentheses.	<p>If x is a macro parameter, the following instances of x as an operand of the # and ## operators do not generate a warning: #x, ##x, and x##. Otherwise, parentheses are required around x.</p> <p>The software does not generate a warning if a parameter is reused as an argument of a function or function-like macro. For example, consider a parameter x. The software does not generate a warning if x appears as (x) or (x, or ,x) or ,x,.</p>
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.	'<name>' is not defined.	
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.	More than one occurrence of the # or ## preprocessor operators.	
19.13	The # and ## preprocessor operators should not be used	Message on definitions of macros using # or ## operators	
19.14	The defined preprocessor operator shall only be used in one of the two standard forms.	'defined' without an identifier.	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.	Precautions shall be taken in order to prevent multiple inclusions.	When a header file is formatted as, <pre>#ifndef <control macro> #define <control macro> <contents> #endif</pre> or, <pre>#ifndef <control macro> #error ... #else #define <control macro> <contents> #endif</pre> it is assumed that precautions have been taken to prevent multiple inclusions. Otherwise, a violation of this MISRA rule is detected.
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	directive is not syntactically meaningful.	
19.17	All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> or <code>#ifdef</code> directive to which they are related.	<ul style="list-style-type: none"> • <code>'#elif'</code> not within a conditional. • <code>'#else'</code> not within a conditional. • <code>'#elif'</code> not within a conditional. • <code>'#endif'</code> not within a conditional. • unbalanced <code>'#endif'</code>. • unterminated <code>'#if'</code> conditional. • unterminated <code>'#ifdef'</code> conditional. • unterminated <code>'#ifndef'</code> conditional. 	

Standard Libraries

N.	MISRA Definition	Messages in report file	Polyspace Implementation
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be redefined. • The macro '<code><name></code>' shall not be undefined. 	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
20.2	The names of standard library macros, objects and functions shall not be reused.	Identifier XX should not be used.	<p>In case a macro whose name corresponds to a standard library macro, object or function is defined, the rule that is detected as violated is 20.1.</p> <p>Tentative definitions are considered as definitions. For objects with file scope, tentative definitions are declarations that:</p> <ul style="list-style-type: none"> • Do not have initializers. • Do not have storage class specifiers, or have the <code>static</code> specifier

N.	MISRA Definition	Messages in report file	Polyspace Implementation
20.3	The validity of values passed to library functions shall be checked.	Validity of values passed to library functions shall be checked	<p>Warning for argument in library function call if the following are all true:</p> <ul style="list-style-type: none"> • Argument is a local variable • Local variable is not tested between last assignment and call to the library function • Library function is a common mathematical function • Corresponding parameter of the library function has a restricted input domain. <p>The library function can be one of the following : sqrt, tan, pow, log, log10, fmod, acos, asin, acosh, atanh, or atan2.</p> <p>You might be using a custom library of mathematical functions. If a custom library function have the same domain and range as another function from the standard library, you can extend this checker to check the custom library function. See “Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries” on page 18-24.</p> <p>A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values” on page 18-26.</p> <p>By default, a Bug Finder analysis does not recognize infinities and NaNs. Operations that results in infinities and NaNs might be flagged as defects. To handle infinities and NaN values in your code, use the option Consider</p>

N.	MISRA Definition	Messages in report file	Polyspace Implementation
			non finite floats (-allow-non-finite-floats).
20.4	Dynamic heap memory allocation shall not be used.	<ul style="list-style-type: none"> • The macro '<name> shall not be used. • Identifier XX should not be used. 	In case the dynamic heap memory allocation functions are actually macros and the macro is expanded in the code, this rule is detected as violated. Assumes rule 20.2 is not violated.
20.5	The error indicator errno shall not be used	The error indicator errno shall not be used	Assumes that rule 20.2 is not violated
20.6	The macro <i>offsetof</i> , in library <stddef.h>, shall not be used.	<ul style="list-style-type: none"> • The macro '<name> shall not be used. • Identifier XX should not be used. 	Assumes that rule 20.2 is not violated
20.7	The <i>setjmp</i> macro and the <i>longjmp</i> function shall not be used.	<ul style="list-style-type: none"> • The macro '<name> shall not be used. • Identifier XX should not be used. 	In case the longjmp function is actually a macro and the macro is expanded in the code, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.8	The signal handling facilities of <signal.h> shall not be used.	<ul style="list-style-type: none"> • The macro '<name> shall not be used. • Identifier XX should not be used. 	In case some of the signal functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.9	The input/output library <stdio.h> shall not be used in production code.	<ul style="list-style-type: none"> • The macro '<name> shall not be used. • Identifier XX should not be used. 	In case the input/output library functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.10	The library functions atof, atoi and atoll from library <stdlib.h> shall not be used.	<ul style="list-style-type: none"> • The macro '<name> shall not be used. • Identifier XX should not be used. 	In case the atof, atoi and atoll functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.11	The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.	<ul style="list-style-type: none"> • The macro '<name> shall not be used. • Identifier XX should not be used. 	In case the abort, exit, getenv and system functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule 20.2 is not violated

N.	MISRA Definition	Messages in report file	Polyspace Implementation
20.12	The time handling functions of library <time.h> shall not be used.	<ul style="list-style-type: none"> The macro '<name>' shall not be used. Identifier XX should not be used. 	In case the time handling functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule 20.2 is not violated

Runtime Failures

N.	MISRA Definition	Messages in report file	Polyspace Implementation
21.1	Minimization of runtime failures shall be ensured by the use of at least one of: <ul style="list-style-type: none"> static verification tools/techniques; dynamic verification tools/techniques; explicit coding of checks to handle runtime faults. 		

Unsupported MISRA C:2004 and MISRA AC AGC Rules

The Polyspace coding rules checker does not check the following MISRA C:2004 coding rules. These rules cannot be enforced because they are outside the scope of Polyspace software. They may concern documentation, dynamic aspects, or functional aspects of MISRA rules. The **Additional Information** column describes the reason each rule is not checked.

Environment

Rule	Description	Additional Information
1.2 (Required)	No reliance shall be placed on undefined or unspecified behavior	Not statically checkable unless the data dynamic properties is taken into account
1.3 (Required)	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the language/compilers/assemblers conform.	It is a process rule method.
1.4 (Required)	The compiler/linker/Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.	To observe this rule, check your compiler documentation.
1.5 (Advisory)	Floating point implementations should comply with a defined floating point standard.	To observe this rule, check your compiler documentation.

Documentation

Rule	Description	Additional Information
3.1 (Required)	All usage of implementation-defined behavior shall be documented.	To observe this rule, check your compiler documentation. Error detection is based on undefined behavior, according to choices made for implementation- defined constructions.
3.2 (Required)	The character set and the corresponding encoding shall be documented.	To observe this rule, check your compiler documentation.
3.3 (Advisory)	The implementation of integer division in the chosen compiler should be determined, documented and taken into account.	To observe this rule, check your compiler documentation.
3.5 (Required)	The implementation-defined behavior and packing of bitfields shall be documented if being relied upon.	To observe this rule, check your compiler documentation.
3.6 (Required)	All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.	To observe this rule, check your compiler documentation.

Structures and Unions

Rule	Description	Additional Information
18.3 (Required)	An area of memory shall not be reused for unrelated purposes.	"purpose" is functional design issue.

Required or Mandatory MISRA C:2012 Rules Supported by Polyspace Bug Finder

The MISRA C:2012 standard classifies the rules that compliant C code must follow as **Required** and **Mandatory**. In total, Polyspace supports 126 out of 126 such rules.

Mandatory Rules

Compliant C code must follow these coding rules. The standard does not permit deviation from these rules. Polyspace supports 16 out of 16 such rules.

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 12.5	The <code>sizeof</code> operator shall not have an operand which is a function parameter declared as "array of type"	MISRA C:2012 Rule 12.5
MISRA C:2012 Rule 13.6	The operand of the <code>sizeof</code> operator shall not contain any expression which has potential side effects	MISRA C:2012 Rule 13.6
MISRA C:2012 Rule 17.3	A function shall not be declared implicitly	MISRA C:2012 Rule 17.3
MISRA C:2012 Rule 17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression	MISRA C:2012 Rule 17.4
MISRA C:2012 Rule 17.6	The declaration of an array parameter shall not contain the static keyword between the []	MISRA C:2012 Rule 17.6
MISRA C:2012 Rule 19.1	An object shall not be assigned or copied to an overlapping object	MISRA C:2012 Rule 19.1
MISRA C:2012 Rule 21.13	Any value passed to a function in <code><ctype.h></code> shall be representable as an unsigned char or be the value EOF	MISRA C:2012 Rule 21.13
MISRA C:2012 Rule 21.17	Use of the string handling function from <code><string.h></code> shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters	MISRA C:2012 Rule 21.17
MISRA C:2012 Rule 21.18	The <code>size_t</code> argument passed to any function in <code><string.h></code> shall have an appropriate value	MISRA C:2012 Rule 21.18

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 21.19	The pointers returned by the Standard Library functions <code>localeconv</code> , <code>getenv</code> , <code>setlocale</code> or <code>strerror</code> shall only be used as if they have pointer to <code>const</code> -qualified type	MISRA C:2012 Rule 21.19
MISRA C:2012 Rule 21.20	The pointer returned by the Standard Library functions <code>asctime</code> , <code>ctime</code> , <code>gmtime</code> , <code>localtime</code> , <code>localeconv</code> , <code>getenv</code> , <code>setlocale</code> or <code>strerror</code> shall not be used following a subsequent call to the same function	MISRA C:2012 Rule 21.20
MISRA C:2012 Rule 22.2	A block of memory shall only be freed if it was allocated by means of a Standard Library function	MISRA C:2012 Rule 22.2
MISRA C:2012 Rule 22.4	There shall be no attempt to write to a stream which has been opened as read-only	MISRA C:2012 Rule 22.4
MISRA C:2012 Rule 22.5	A pointer to a FILE object shall not be dereferenced	MISRA C:2012 Rule 22.5
MISRA C:2012 Rule 22.6	The value of a pointer to a FILE shall not be used after the associated stream has been closed	MISRA C:2012 Rule 22.6
MISRA C:2012 Rule 9.1	The value of an object with automatic storage duration shall not be read before it has been set	MISRA C:2012 Rule 9.1

Required Rules

Compliant C code must follow these coding rules. The standard permits only the deviations that you formally record and authorize. Polyspace supports 110 out of 110 such rules.

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 10.1	Operands shall not be of an inappropriate essential type	MISRA C:2012 Rule 10.1
MISRA C:2012 Rule 10.2	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations	MISRA C:2012 Rule 10.2

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category	MISRA C:2012 Rule 10.3
MISRA C:2012 Rule 10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category	MISRA C:2012 Rule 10.4
MISRA C:2012 Rule 10.6	The value of a composite expression shall not be assigned to an object with wider essential type	MISRA C:2012 Rule 10.6
MISRA C:2012 Rule 10.7	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type	MISRA C:2012 Rule 10.7
MISRA C:2012 Rule 10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type	MISRA C:2012 Rule 10.8
MISRA C:2012 Rule 11.1	Conversions shall not be performed between a pointer to a function and any other type	MISRA C:2012 Rule 11.1
MISRA C:2012 Rule 11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type	MISRA C:2012 Rule 11.2
MISRA C:2012 Rule 11.3	A cast shall not be performed between a pointer to object type and a pointer to a different object type	MISRA C:2012 Rule 11.3
MISRA C:2012 Rule 11.6	A cast shall not be performed between pointer to void and an arithmetic type	MISRA C:2012 Rule 11.6
MISRA C:2012 Rule 11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type	MISRA C:2012 Rule 11.7
MISRA C:2012 Rule 11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer	MISRA C:2012 Rule 11.8

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 11.9	The macro NULL shall be the only permitted form of integer null pointer constant	MISRA C:2012 Rule 11.9
MISRA C:2012 Rule 12.2	The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand	MISRA C:2012 Rule 12.2
MISRA C:2012 Rule 13.1	Initializer lists shall not contain persistent side effects	MISRA C:2012 Rule 13.1
MISRA C:2012 Rule 13.2	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders	MISRA C:2012 Rule 13.2
MISRA C:2012 Rule 13.5	The right hand operand of a logical && or operator shall not contain persistent side effects	MISRA C:2012 Rule 13.5
MISRA C:2012 Rule 14.1	A loop counter shall not have essentially floating type	MISRA C:2012 Rule 14.1
MISRA C:2012 Rule 14.2	A for loop shall be well-formed	MISRA C:2012 Rule 14.2
MISRA C:2012 Rule 14.3	Controlling expressions shall not be invariant	MISRA C:2012 Rule 14.3
MISRA C:2012 Rule 14.4	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type	MISRA C:2012 Rule 14.4
MISRA C:2012 Rule 15.2	The goto statement shall jump to a label declared later in the same function	MISRA C:2012 Rule 15.2
MISRA C:2012 Rule 15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement	MISRA C:2012 Rule 15.3
MISRA C:2012 Rule 15.6	The body of an iteration-statement or a selection-statement shall be a compound statement	MISRA C:2012 Rule 15.6
MISRA C:2012 Rule 15.7	All if ... else if constructs shall be terminated with an else statement	MISRA C:2012 Rule 15.7
MISRA C:2012 Rule 16.1	All switch statements shall be well-formed	MISRA C:2012 Rule 16.1

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 16.2	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	MISRA C:2012 Rule 16.2
MISRA C:2012 Rule 16.3	An unconditional break statement shall terminate every switch-clause	MISRA C:2012 Rule 16.3
MISRA C:2012 Rule 16.4	Every switch statement shall have a default label	MISRA C:2012 Rule 16.4
MISRA C:2012 Rule 16.5	A default label shall appear as either the first or the last switch label of a switch statement	MISRA C:2012 Rule 16.5
MISRA C:2012 Rule 16.6	Every switch statement shall have at least two switch-clauses	MISRA C:2012 Rule 16.6
MISRA C:2012 Rule 16.7	A switch-expression shall not have essentially Boolean type	MISRA C:2012 Rule 16.7
MISRA C:2012 Rule 17.1	The features of <stdarg.h> shall not be used	MISRA C:2012 Rule 17.1
MISRA C:2012 Rule 17.2	Functions shall not call themselves, either directly or indirectly	MISRA C:2012 Rule 17.2
MISRA C:2012 Rule 17.7	The value returned by a function having non-void return type shall be used	MISRA C:2012 Rule 17.7
MISRA C:2012 Rule 18.1	A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand	MISRA C:2012 Rule 18.1
MISRA C:2012 Rule 18.2	Subtraction between pointers shall only be applied to pointers that address elements of the same array	MISRA C:2012 Rule 18.2
MISRA C:2012 Rule 18.3	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object	MISRA C:2012 Rule 18.3
MISRA C:2012 Rule 18.6	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist	MISRA C:2012 Rule 18.6
MISRA C:2012 Rule 18.7	Flexible array members shall not be declared	MISRA C:2012 Rule 18.7

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 18.8	Variable-length array types shall not be used	MISRA C:2012 Rule 18.8
MISRA C:2012 Rule 1.1	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits	MISRA C:2012 Rule 1.1
MISRA C:2012 Rule 1.3	There shall be no occurrence of undefined or critical unspecified behaviour	MISRA C:2012 Rule 1.3
MISRA C:2012 Rule 1.4	Emergent language features shall not be used	MISRA C:2012 Rule 1.4
MISRA C:2012 Rule 20.11	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator	MISRA C:2012 Rule 20.11
MISRA C:2012 Rule 20.12	A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators	MISRA C:2012 Rule 20.12
MISRA C:2012 Rule 20.13	A line whose first token is # shall be a valid preprocessing directive	MISRA C:2012 Rule 20.13
MISRA C:2012 Rule 20.14	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related	MISRA C:2012 Rule 20.14
MISRA C:2012 Rule 20.2	The ', " or \ characters and the /* or // character sequences shall not occur in a header file name	MISRA C:2012 Rule 20.2
MISRA C:2012 Rule 20.3	The #include directive shall be followed by either a <filename> or "filename" sequence	MISRA C:2012 Rule 20.3
MISRA C:2012 Rule 20.4	A macro shall not be defined with the same name as a keyword	MISRA C:2012 Rule 20.4
MISRA C:2012 Rule 20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument	MISRA C:2012 Rule 20.6

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses	MISRA C:2012 Rule 20.7
MISRA C:2012 Rule 20.8	The controlling expression of a <code>#if</code> or <code>#elif</code> preprocessing directive shall evaluate to 0 or 1	MISRA C:2012 Rule 20.8
MISRA C:2012 Rule 20.9	All identifiers used in the controlling expression of <code>#if</code> or <code>#elif</code> preprocessing directives shall be <code>#define'd</code> before evaluation	MISRA C:2012 Rule 20.9
MISRA C:2012 Rule 21.1	<code>#define</code> and <code>#undef</code> shall not be used on a reserved identifier or reserved macro name	MISRA C:2012 Rule 21.1
MISRA C:2012 Rule 21.10	The Standard Library time and date functions shall not be used	MISRA C:2012 Rule 21.10
MISRA C:2012 Rule 21.11	The standard header file <code><tgmath.h></code> shall not be used	MISRA C:2012 Rule 21.11
MISRA C:2012 Rule 21.14	The Standard Library function <code>memcmp</code> shall not be used to compare null terminated strings	MISRA C:2012 Rule 21.14
MISRA C:2012 Rule 21.15	The pointer arguments to the Standard Library functions <code>memcpy</code> , <code>memmove</code> and <code>memcmp</code> shall be pointers to qualified or unqualified versions of compatible types	MISRA C:2012 Rule 21.15
MISRA C:2012 Rule 21.16	The pointer arguments to the Standard Library function <code>memcmp</code> shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type	MISRA C:2012 Rule 21.16
MISRA C:2012 Rule 21.2	A reserved identifier or reserved macro name shall not be declared	MISRA C:2012 Rule 21.2
MISRA C:2012 Rule 21.21	The Standard Library function <code>system</code> of <code><stdlib.h></code> shall not be used	MISRA C:2012 Rule 21.21
MISRA C:2012 Rule 21.3	The memory allocation and deallocation functions of <code><stdlib.h></code> shall not be used	MISRA C:2012 Rule 21.3

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 21.4	The standard header file <setjmp.h> shall not be used	MISRA C:2012 Rule 21.4
MISRA C:2012 Rule 21.5	The standard header file <signal.h> shall not be used	MISRA C:2012 Rule 21.5
MISRA C:2012 Rule 21.6	The Standard Library input/output functions shall not be used	MISRA C:2012 Rule 21.6
MISRA C:2012 Rule 21.7	The Standard Library functions <code>atof</code> , <code>atoi</code> , <code>atol</code> , and <code>atoll</code> functions of <stdlib.h> shall not be used	MISRA C:2012 Rule 21.7
MISRA C:2012 Rule 21.8	The Standard Library termination functions of <stdlib.h> shall not be used	MISRA C:2012 Rule 21.8
MISRA C:2012 Rule 21.9	The Standard Library library functions <code>bsearch</code> and <code>qsort</code> of <stdlib.h> shall not be used	MISRA C:2012 Rule 21.9
MISRA C:2012 Rule 22.1	All resources obtained dynamically by means of Standard Library functions shall be explicitly released	MISRA C:2012 Rule 22.1
MISRA C:2012 Rule 22.10	The value of <code>errno</code> shall only be tested when the last function to be called was an <code>errno</code> -setting function	MISRA C:2012 Rule 22.10
MISRA C:2012 Rule 22.3	The same file shall not be open for read and write access at the same time on different streams	MISRA C:2012 Rule 22.3
MISRA C:2012 Rule 22.7	The macro <code>EOF</code> shall only be compared with the unmodified return value from any Standard Library function capable of returning <code>EOF</code>	MISRA C:2012 Rule 22.7
MISRA C:2012 Rule 22.8	The value of <code>errno</code> shall be set to zero prior to a call to an <code>errno</code> -setting-function	MISRA C:2012 Rule 22.8
MISRA C:2012 Rule 22.9	The value of <code>errno</code> shall be tested against zero after calling an <code>errno</code> -setting function	MISRA C:2012 Rule 22.9
MISRA C:2012 Rule 2.1	A project shall not contain unreachable code	MISRA C:2012 Rule 2.1
MISRA C:2012 Rule 2.2	There shall be no dead code	MISRA C:2012 Rule 2.2

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 3.1	The character sequences /* and // shall not be used within a comment	MISRA C:2012 Rule 3.1
MISRA C:2012 Rule 3.2	Line-splicing shall not be used in // comments	MISRA C:2012 Rule 3.2
MISRA C:2012 Rule 4.1	Octal and hexadecimal escape sequences shall be terminated	MISRA C:2012 Rule 4.1
MISRA C:2012 Rule 5.1	External identifiers shall be distinct	MISRA C:2012 Rule 5.1
MISRA C:2012 Rule 5.2	Identifiers declared in the same scope and name space shall be distinct	MISRA C:2012 Rule 5.2
MISRA C:2012 Rule 5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope	MISRA C:2012 Rule 5.3
MISRA C:2012 Rule 5.4	Macro identifiers shall be distinct	MISRA C:2012 Rule 5.4
MISRA C:2012 Rule 5.5	Identifiers shall be distinct from macro names	MISRA C:2012 Rule 5.5
MISRA C:2012 Rule 5.6	A typedef name shall be a unique identifier	MISRA C:2012 Rule 5.6
MISRA C:2012 Rule 5.7	A tag name shall be a unique identifier	MISRA C:2012 Rule 5.7
MISRA C:2012 Rule 5.8	Identifiers that define objects or functions with external linkage shall be unique	MISRA C:2012 Rule 5.8
MISRA C:2012 Rule 6.1	Bit-fields shall only be declared with an appropriate type	MISRA C:2012 Rule 6.1
MISRA C:2012 Rule 6.2	Single-bit named bit fields shall not be of a signed type	MISRA C:2012 Rule 6.2
MISRA C:2012 Rule 7.1	Octal constants shall not be used	MISRA C:2012 Rule 7.1
MISRA C:2012 Rule 7.2	A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type	MISRA C:2012 Rule 7.2
MISRA C:2012 Rule 7.3	The lowercase character “l” shall not be used in a literal suffix	MISRA C:2012 Rule 7.3
MISRA C:2012 Rule 7.4	A string literal shall not be assigned to an object unless the object’s type is “pointer to const-qualified char”	MISRA C:2012 Rule 7.4

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 8.1	Types shall be explicitly specified	MISRA C:2012 Rule 8.1
MISRA C:2012 Rule 8.10	An inline function shall be declared with the static storage class	MISRA C:2012 Rule 8.10
MISRA C:2012 Rule 8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique	MISRA C:2012 Rule 8.12
MISRA C:2012 Rule 8.14	The restrict type qualifier shall not be used	MISRA C:2012 Rule 8.14
MISRA C:2012 Rule 8.2	Function types shall be in prototype form with named parameters	MISRA C:2012 Rule 8.2
MISRA C:2012 Rule 8.3	All declarations of an object or function shall use the same names and type qualifiers	MISRA C:2012 Rule 8.3
MISRA C:2012 Rule 8.4	A compatible declaration shall be visible when an object or function with external linkage is defined	MISRA C:2012 Rule 8.4
MISRA C:2012 Rule 8.5	An external object or function shall be declared once in one and only one file	MISRA C:2012 Rule 8.5
MISRA C:2012 Rule 8.6	An identifier with external linkage shall have exactly one external definition	MISRA C:2012 Rule 8.6
MISRA C:2012 Rule 8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage	MISRA C:2012 Rule 8.8
MISRA C:2012 Rule 9.2	The initializer for an aggregate or union shall be enclosed in braces	MISRA C:2012 Rule 9.2
MISRA C:2012 Rule 9.3	Arrays shall not be partially initialized	MISRA C:2012 Rule 9.3
MISRA C:2012 Rule 9.4	An element of an object shall not be initialized more than once	MISRA C:2012 Rule 9.4
MISRA C:2012 Rule 9.5	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly	MISRA C:2012 Rule 9.5

See Also

Check MISRA C:2012 (-misra3)

More About

- “Check for and Review Coding Standard Violations” on page 16-2
- “Coding Standards”
- “Decidable MISRA C:2012 Rules Supported by Polyspace Bug Finder” on page 17-54
- “Undecidable MISRA C:2012 Rules and Directives Supported by Polyspace Bug Finder” on page 17-64
- “Required and Statically Enforceable CERT C Rules Supported by Polyspace Bug Finder” on page 17-75
- “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78

Decidable MISRA C:2012 Rules Supported by Polyspace Bug Finder

The MISRA C:2012 standard classifies rules that can be statically enforced in all possible cases as **Decidable**. Polyspace supports 122 out of 122 such rules. None of the MISRA C:2012 directives are statically enforceable.

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 10.1	Operands shall not be of an inappropriate essential type	MISRA C:2012 Rule 10.1
MISRA C:2012 Rule 10.2	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations	MISRA C:2012 Rule 10.2
MISRA C:2012 Rule 10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category	MISRA C:2012 Rule 10.3
MISRA C:2012 Rule 10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category	MISRA C:2012 Rule 10.4
MISRA C:2012 Rule 10.5	The value of an expression should not be cast to an inappropriate essential type	MISRA C:2012 Rule 10.5
MISRA C:2012 Rule 10.6	The value of a composite expression shall not be assigned to an object with wider essential type	MISRA C:2012 Rule 10.6
MISRA C:2012 Rule 10.7	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type	MISRA C:2012 Rule 10.7
MISRA C:2012 Rule 10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type	MISRA C:2012 Rule 10.8
MISRA C:2012 Rule 11.1	Conversions shall not be performed between a pointer to a function and any other type	MISRA C:2012 Rule 11.1

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type	MISRA C:2012 Rule 11.2
MISRA C:2012 Rule 11.3	A cast shall not be performed between a pointer to object type and a pointer to a different object type	MISRA C:2012 Rule 11.3
MISRA C:2012 Rule 11.4	A conversion should not be performed between a pointer to object and an integer type	MISRA C:2012 Rule 11.4
MISRA C:2012 Rule 11.5	A conversion should not be performed from pointer to void into pointer to object	MISRA C:2012 Rule 11.5
MISRA C:2012 Rule 11.6	A cast shall not be performed between pointer to void and an arithmetic type	MISRA C:2012 Rule 11.6
MISRA C:2012 Rule 11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type	MISRA C:2012 Rule 11.7
MISRA C:2012 Rule 11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer	MISRA C:2012 Rule 11.8
MISRA C:2012 Rule 11.9	The macro NULL shall be the only permitted form of integer null pointer constant	MISRA C:2012 Rule 11.9
MISRA C:2012 Rule 12.1	The precedence of operators within expressions should be made explicit	MISRA C:2012 Rule 12.1
MISRA C:2012 Rule 12.3	The comma operator should not be used	MISRA C:2012 Rule 12.3
MISRA C:2012 Rule 12.4	Evaluation of constant expressions should not lead to unsigned integer wrap-around	MISRA C:2012 Rule 12.4
MISRA C:2012 Rule 12.5	The sizeof operator shall not have an operand which is a function parameter declared as "array of type"	MISRA C:2012 Rule 12.5

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 13.3	A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator	MISRA C:2012 Rule 13.3
MISRA C:2012 Rule 13.4	The result of an assignment operator should not be used	MISRA C:2012 Rule 13.4
MISRA C:2012 Rule 13.6	The operand of the sizeof operator shall not contain any expression which has potential side effects	MISRA C:2012 Rule 13.6
MISRA C:2012 Rule 14.4	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type	MISRA C:2012 Rule 14.4
MISRA C:2012 Rule 15.1	The goto statement should not be used	MISRA C:2012 Rule 15.1
MISRA C:2012 Rule 15.2	The goto statement shall jump to a label declared later in the same function	MISRA C:2012 Rule 15.2
MISRA C:2012 Rule 15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement	MISRA C:2012 Rule 15.3
MISRA C:2012 Rule 15.4	There should be no more than one break or goto statement used to terminate any iteration statement	MISRA C:2012 Rule 15.4
MISRA C:2012 Rule 15.5	A function should have a single point of exit at the end	MISRA C:2012 Rule 15.5
MISRA C:2012 Rule 15.6	The body of an iteration-statement or a selection-statement shall be a compound statement	MISRA C:2012 Rule 15.6
MISRA C:2012 Rule 15.7	All if ... else if constructs shall be terminated with an else statement	MISRA C:2012 Rule 15.7
MISRA C:2012 Rule 16.1	All switch statements shall be well-formed	MISRA C:2012 Rule 16.1

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 16.2	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	MISRA C:2012 Rule 16.2
MISRA C:2012 Rule 16.3	An unconditional break statement shall terminate every switch-clause	MISRA C:2012 Rule 16.3
MISRA C:2012 Rule 16.4	Every switch statement shall have a default label	MISRA C:2012 Rule 16.4
MISRA C:2012 Rule 16.5	A default label shall appear as either the first or the last switch label of a switch statement	MISRA C:2012 Rule 16.5
MISRA C:2012 Rule 16.6	Every switch statement shall have at least two switch-clauses	MISRA C:2012 Rule 16.6
MISRA C:2012 Rule 16.7	A switch-expression shall not have essentially Boolean type	MISRA C:2012 Rule 16.7
MISRA C:2012 Rule 17.1	The features of <stdarg.h> shall not be used	MISRA C:2012 Rule 17.1
MISRA C:2012 Rule 17.3	A function shall not be declared implicitly	MISRA C:2012 Rule 17.3
MISRA C:2012 Rule 17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression	MISRA C:2012 Rule 17.4
MISRA C:2012 Rule 17.6	The declaration of an array parameter shall not contain the static keyword between the []	MISRA C:2012 Rule 17.6
MISRA C:2012 Rule 17.7	The value returned by a function having non-void return type shall be used	MISRA C:2012 Rule 17.7
MISRA C:2012 Rule 18.4	The +, -, += and -= operators should not be applied to an expression of pointer type	MISRA C:2012 Rule 18.4
MISRA C:2012 Rule 18.5	Declarations should contain no more than two levels of pointer nesting	MISRA C:2012 Rule 18.5
MISRA C:2012 Rule 18.7	Flexible array members shall not be declared	MISRA C:2012 Rule 18.7
MISRA C:2012 Rule 18.8	Variable-length array types shall not be used	MISRA C:2012 Rule 18.8
MISRA C:2012 Rule 19.2	The union keyword should not be used	MISRA C:2012 Rule 19.2

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 1.1	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits	MISRA C:2012 Rule 1.1
MISRA C:2012 Rule 1.2	Language extensions should not be used	MISRA C:2012 Rule 1.2
MISRA C:2012 Rule 1.4	Emergent language features shall not be used	MISRA C:2012 Rule 1.4
MISRA C:2012 Rule 20.1	#include directives should only be preceded by preprocessor directives or comments	MISRA C:2012 Rule 20.1
MISRA C:2012 Rule 20.10	The # and ## preprocessor operators should not be used	MISRA C:2012 Rule 20.10
MISRA C:2012 Rule 20.11	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator	MISRA C:2012 Rule 20.11
MISRA C:2012 Rule 20.12	A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators	MISRA C:2012 Rule 20.12
MISRA C:2012 Rule 20.13	A line whose first token is # shall be a valid preprocessing directive	MISRA C:2012 Rule 20.13
MISRA C:2012 Rule 20.14	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related	MISRA C:2012 Rule 20.14
MISRA C:2012 Rule 20.2	The ', " or \ characters and the /* or // character sequences shall not occur in a header file name	MISRA C:2012 Rule 20.2
MISRA C:2012 Rule 20.3	The #include directive shall be followed by either a <filename> or "filename" sequence	MISRA C:2012 Rule 20.3
MISRA C:2012 Rule 20.4	A macro shall not be defined with the same name as a keyword	MISRA C:2012 Rule 20.4
MISRA C:2012 Rule 20.5	#undef should not be used	MISRA C:2012 Rule 20.5

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument	MISRA C:2012 Rule 20.6
MISRA C:2012 Rule 20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses	MISRA C:2012 Rule 20.7
MISRA C:2012 Rule 20.8	The controlling expression of a <code>#if</code> or <code>#elif</code> preprocessing directive shall evaluate to 0 or 1	MISRA C:2012 Rule 20.8
MISRA C:2012 Rule 20.9	All identifiers used in the controlling expression of <code>#if</code> or <code>#elif</code> preprocessing directives shall be <code>#define'd</code> before evaluation	MISRA C:2012 Rule 20.9
MISRA C:2012 Rule 21.1	<code>#define</code> and <code>#undef</code> shall not be used on a reserved identifier or reserved macro name	MISRA C:2012 Rule 21.1
MISRA C:2012 Rule 21.10	The Standard Library time and date functions shall not be used	MISRA C:2012 Rule 21.10
MISRA C:2012 Rule 21.11	The standard header file <code><tgmath.h></code> shall not be used	MISRA C:2012 Rule 21.11
MISRA C:2012 Rule 21.12	The exception handling features of <code><fenv.h></code> should not be used	MISRA C:2012 Rule 21.12
MISRA C:2012 Rule 21.15	The pointer arguments to the Standard Library functions <code>memcpy</code> , <code>memmove</code> and <code>memcmp</code> shall be pointers to qualified or unqualified versions of compatible types	MISRA C:2012 Rule 21.15
MISRA C:2012 Rule 21.16	The pointer arguments to the Standard Library function <code>memcmp</code> shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type	MISRA C:2012 Rule 21.16
MISRA C:2012 Rule 21.2	A reserved identifier or reserved macro name shall not be declared	MISRA C:2012 Rule 21.2
MISRA C:2012 Rule 21.21	The Standard Library function system of <code><stdlib.h></code> shall not be used	MISRA C:2012 Rule 21.21

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 21.3	The memory allocation and deallocation functions of <code><stdlib.h></code> shall not be used	MISRA C:2012 Rule 21.3
MISRA C:2012 Rule 21.4	The standard header file <code><setjmp.h></code> shall not be used	MISRA C:2012 Rule 21.4
MISRA C:2012 Rule 21.5	The standard header file <code><signal.h></code> shall not be used	MISRA C:2012 Rule 21.5
MISRA C:2012 Rule 21.6	The Standard Library input/output functions shall not be used	MISRA C:2012 Rule 21.6
MISRA C:2012 Rule 21.7	The Standard Library functions <code>atof</code> , <code>atoi</code> , <code>atol</code> , and <code>atoll</code> functions of <code><stdlib.h></code> shall not be used	MISRA C:2012 Rule 21.7
MISRA C:2012 Rule 21.8	The Standard Library termination functions of <code><stdlib.h></code> shall not be used	MISRA C:2012 Rule 21.8
MISRA C:2012 Rule 21.9	The Standard Library library functions <code>bsearch</code> and <code>qsort</code> of <code><stdlib.h></code> shall not be used	MISRA C:2012 Rule 21.9
MISRA C:2012 Rule 2.3	A project should not contain unused type declarations	MISRA C:2012 Rule 2.3
MISRA C:2012 Rule 2.4	A project should not contain unused tag declarations	MISRA C:2012 Rule 2.4
MISRA C:2012 Rule 2.5	A project should not contain unused macro declarations	MISRA C:2012 Rule 2.5
MISRA C:2012 Rule 2.6	A function should not contain unused label declarations	MISRA C:2012 Rule 2.6
MISRA C:2012 Rule 2.7	There should be no unused parameters in functions	MISRA C:2012 Rule 2.7
MISRA C:2012 Rule 3.1	The character sequences <code>/*</code> and <code>//</code> shall not be used within a comment	MISRA C:2012 Rule 3.1
MISRA C:2012 Rule 3.2	Line-splicing shall not be used in <code>//</code> comments	MISRA C:2012 Rule 3.2
MISRA C:2012 Rule 4.1	Octal and hexadecimal escape sequences shall be terminated	MISRA C:2012 Rule 4.1
MISRA C:2012 Rule 4.2	Trigraphs should not be used	MISRA C:2012 Rule 4.2
MISRA C:2012 Rule 5.1	External identifiers shall be distinct	MISRA C:2012 Rule 5.1
MISRA C:2012 Rule 5.2	Identifiers declared in the same scope and name space shall be distinct	MISRA C:2012 Rule 5.2

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope	MISRA C:2012 Rule 5.3
MISRA C:2012 Rule 5.4	Macro identifiers shall be distinct	MISRA C:2012 Rule 5.4
MISRA C:2012 Rule 5.5	Identifiers shall be distinct from macro names	MISRA C:2012 Rule 5.5
MISRA C:2012 Rule 5.6	A typedef name shall be a unique identifier	MISRA C:2012 Rule 5.6
MISRA C:2012 Rule 5.7	A tag name shall be a unique identifier	MISRA C:2012 Rule 5.7
MISRA C:2012 Rule 5.8	Identifiers that define objects or functions with external linkage shall be unique	MISRA C:2012 Rule 5.8
MISRA C:2012 Rule 5.9	Identifiers that define objects or functions with internal linkage should be unique	MISRA C:2012 Rule 5.9
MISRA C:2012 Rule 6.1	Bit-fields shall only be declared with an appropriate type	MISRA C:2012 Rule 6.1
MISRA C:2012 Rule 6.2	Single-bit named bit fields shall not be of a signed type	MISRA C:2012 Rule 6.2
MISRA C:2012 Rule 7.1	Octal constants shall not be used	MISRA C:2012 Rule 7.1
MISRA C:2012 Rule 7.2	A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type	MISRA C:2012 Rule 7.2
MISRA C:2012 Rule 7.3	The lowercase character “l” shall not be used in a literal suffix	MISRA C:2012 Rule 7.3
MISRA C:2012 Rule 7.4	A string literal shall not be assigned to an object unless the object’s type is “pointer to const-qualified char”	MISRA C:2012 Rule 7.4
MISRA C:2012 Rule 8.1	Types shall be explicitly specified	MISRA C:2012 Rule 8.1
MISRA C:2012 Rule 8.10	An inline function shall be declared with the static storage class	MISRA C:2012 Rule 8.10
MISRA C:2012 Rule 8.11	When an array with external linkage is declared, its size should be explicitly specified	MISRA C:2012 Rule 8.11

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique	MISRA C:2012 Rule 8.12
MISRA C:2012 Rule 8.14	The restrict type qualifier shall not be used	MISRA C:2012 Rule 8.14
MISRA C:2012 Rule 8.2	Function types shall be in prototype form with named parameters	MISRA C:2012 Rule 8.2
MISRA C:2012 Rule 8.3	All declarations of an object or function shall use the same names and type qualifiers	MISRA C:2012 Rule 8.3
MISRA C:2012 Rule 8.4	A compatible declaration shall be visible when an object or function with external linkage is defined	MISRA C:2012 Rule 8.4
MISRA C:2012 Rule 8.5	An external object or function shall be declared once in one and only one file	MISRA C:2012 Rule 8.5
MISRA C:2012 Rule 8.6	An identifier with external linkage shall have exactly one external definition	MISRA C:2012 Rule 8.6
MISRA C:2012 Rule 8.7	Functions and objects should not be defined with external linkage if they are referenced in only one translation unit	MISRA C:2012 Rule 8.7
MISRA C:2012 Rule 8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage	MISRA C:2012 Rule 8.8
MISRA C:2012 Rule 8.9	An object should be defined at block scope if its identifier only appears in a single function	MISRA C:2012 Rule 8.9
MISRA C:2012 Rule 9.2	The initializer for an aggregate or union shall be enclosed in braces	MISRA C:2012 Rule 9.2
MISRA C:2012 Rule 9.3	Arrays shall not be partially initialized	MISRA C:2012 Rule 9.3
MISRA C:2012 Rule 9.4	An element of an object shall not be initialized more than once	MISRA C:2012 Rule 9.4
MISRA C:2012 Rule 9.5	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly	MISRA C:2012 Rule 9.5

See Also

Check MISRA C:2012 (-misra3)

More About

- “Check for and Review Coding Standard Violations” on page 16-2
- “Coding Standards”
- “Required or Mandatory MISRA C:2012 Rules Supported by Polyspace Bug Finder” on page 17-43
- “Required and Statically Enforceable CERT C Rules Supported by Polyspace Bug Finder” on page 17-75
- “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78
- “Undecidable MISRA C:2012 Rules and Directives Supported by Polyspace Bug Finder” on page 17-64

Undecidable MISRA C:2012 Rules and Directives Supported by Polyspace Bug Finder

The MISRA C:2012 standard classifies rules and directives that cannot be statically enforced in every possible cases as **Undecidable**. Polyspace supports 36 out of 36 such rules, and 15 out of 17 such directives.

Undecidable Rules

A rule is undecidable if a static analysis tool can check compliance to it only in certain cases. Polyspace shows the subset of all possible issues. For details about which issues Polyspace detects for a particular rule, see the **Polyspace Implementation** section in the reference page of the rule. Polyspace supports 36 out of 36 such rules.

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 12.2	The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand	MISRA C:2012 Rule 12.2
MISRA C:2012 Rule 13.1	Initializer lists shall not contain persistent side effects	MISRA C:2012 Rule 13.1
MISRA C:2012 Rule 13.2	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders	MISRA C:2012 Rule 13.2
MISRA C:2012 Rule 13.5	The right hand operand of a logical && or operator shall not contain persistent side effects	MISRA C:2012 Rule 13.5
MISRA C:2012 Rule 14.1	A loop counter shall not have essentially floating type	MISRA C:2012 Rule 14.1
MISRA C:2012 Rule 14.2	A for loop shall be well-formed	MISRA C:2012 Rule 14.2
MISRA C:2012 Rule 14.3	Controlling expressions shall not be invariant	MISRA C:2012 Rule 14.3
MISRA C:2012 Rule 17.2	Functions shall not call themselves, either directly or indirectly	MISRA C:2012 Rule 17.2
MISRA C:2012 Rule 17.5	The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements	MISRA C:2012 Rule 17.5
MISRA C:2012 Rule 17.8	A function parameter should not be modified	MISRA C:2012 Rule 17.8

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 18.1	A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand	MISRA C:2012 Rule 18.1
MISRA C:2012 Rule 18.2	Subtraction between pointers shall only be applied to pointers that address elements of the same array	MISRA C:2012 Rule 18.2
MISRA C:2012 Rule 18.3	The relational operators <code>></code> , <code>>=</code> , <code><</code> and <code><=</code> shall not be applied to objects of pointer type except where they point into the same object	MISRA C:2012 Rule 18.3
MISRA C:2012 Rule 18.6	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist	MISRA C:2012 Rule 18.6
MISRA C:2012 Rule 19.1	An object shall not be assigned or copied to an overlapping object	MISRA C:2012 Rule 19.1
MISRA C:2012 Rule 1.3	There shall be no occurrence of undefined or critical unspecified behaviour	MISRA C:2012 Rule 1.3
MISRA C:2012 Rule 21.13	Any value passed to a function in <code><ctype.h></code> shall be representable as an unsigned char or be the value EOF	MISRA C:2012 Rule 21.13
MISRA C:2012 Rule 21.14	The Standard Library function <code>memcmp</code> shall not be used to compare null terminated strings	MISRA C:2012 Rule 21.14
MISRA C:2012 Rule 21.17	Use of the string handling function from <code><string.h></code> shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters	MISRA C:2012 Rule 21.17
MISRA C:2012 Rule 21.18	The <code>size_t</code> argument passed to any function in <code><string.h></code> shall have an appropriate value	MISRA C:2012 Rule 21.18

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 21.19	The pointers returned by the Standard Library functions <code>localeconv</code> , <code>getenv</code> , <code>setlocale</code> or <code>strerror</code> shall only be used as if they have pointer to <code>const</code> -qualified type	MISRA C:2012 Rule 21.19
MISRA C:2012 Rule 21.20	The pointer returned by the Standard Library functions <code>asctime</code> , <code>ctime</code> , <code>gmtime</code> , <code>localtime</code> , <code>localeconv</code> , <code>getenv</code> , <code>setlocale</code> or <code>strerror</code> shall not be used following a subsequent call to the same function	MISRA C:2012 Rule 21.20
MISRA C:2012 Rule 22.1	All resources obtained dynamically by means of Standard Library functions shall be explicitly released	MISRA C:2012 Rule 22.1
MISRA C:2012 Rule 22.10	The value of <code>errno</code> shall only be tested when the last function to be called was an <code>errno</code> -setting function	MISRA C:2012 Rule 22.10
MISRA C:2012 Rule 22.2	A block of memory shall only be freed if it was allocated by means of a Standard Library function	MISRA C:2012 Rule 22.2
MISRA C:2012 Rule 22.3	The same file shall not be open for read and write access at the same time on different streams	MISRA C:2012 Rule 22.3
MISRA C:2012 Rule 22.4	There shall be no attempt to write to a stream which has been opened as read-only	MISRA C:2012 Rule 22.4
MISRA C:2012 Rule 22.5	A pointer to a <code>FILE</code> object shall not be dereferenced	MISRA C:2012 Rule 22.5
MISRA C:2012 Rule 22.6	The value of a pointer to a <code>FILE</code> shall not be used after the associated stream has been closed	MISRA C:2012 Rule 22.6
MISRA C:2012 Rule 22.7	The macro <code>EOF</code> shall only be compared with the unmodified return value from any Standard Library function capable of returning <code>EOF</code>	MISRA C:2012 Rule 22.7
MISRA C:2012 Rule 22.8	The value of <code>errno</code> shall be set to zero prior to a call to an <code>errno</code> -setting-function	MISRA C:2012 Rule 22.8

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 22.9	The value of <code>errno</code> shall be tested against zero after calling an <code>errno</code> -setting function	MISRA C:2012 Rule 22.9
MISRA C:2012 Rule 2.1	A project shall not contain unreachable code	MISRA C:2012 Rule 2.1
MISRA C:2012 Rule 2.2	There shall be no dead code	MISRA C:2012 Rule 2.2
MISRA C:2012 Rule 8.13	A pointer should point to a const-qualified type whenever possible	MISRA C:2012 Rule 8.13
MISRA C:2012 Rule 9.1	The value of an object with automatic storage duration shall not be read before it has been set	MISRA C:2012 Rule 9.1

Undecidable Directives

A directive is undecidable if a static analysis tool can check compliance to it only in certain cases. Polyspace shows the subset of all possible issues. For details about which issues Polyspace detects for a particular directive, see the **Polyspace Implementation** section in the reference page of the directive. Polyspace supports 15 out of 17 such directives.

MISRA C:2012 Directives	Description	Polyspace Checker
MISRA C:2012 Dir 1.1	Any implementation-defined behavior on which the output of the program depends shall be documented and understood	MISRA C:2012 Dir 1.1
MISRA C:2012 Dir 2.1	All source files shall compile without any compilation errors	MISRA C:2012 Dir 2.1
MISRA C:2012 Dir 4.1	Run-time failures shall be minimized	MISRA C:2012 Dir 4.1
MISRA C:2012 Dir 4.10	Precautions shall be taken in order to prevent the contents of a header file being included more than once	MISRA C:2012 Dir 4.10
MISRA C:2012 Dir 4.11	The validity of values passed to library functions shall be checked	MISRA C:2012 Dir 4.11
MISRA C:2012 Dir 4.12	Dynamic memory allocation shall not be used	MISRA C:2012 Dir 4.12
MISRA C:2012 Dir 4.13	Functions which are designed to provide operations on a resource should be called in an appropriate sequence	MISRA C:2012 Dir 4.13

MISRA C:2012 Directives	Description	Polyspace Checker
MISRA C:2012 Dir 4.14	The validity of values received from external sources shall be checked	MISRA C:2012 Dir 4.14
MISRA C:2012 Dir 4.3	Assembly language shall be encapsulated and isolated	MISRA C:2012 Dir 4.3
MISRA C:2012 Dir 4.4	Sections of code should not be "commented out"	MISRA C:2012 Dir 4.4
MISRA C:2012 Dir 4.5	Identifiers in the same name space with overlapping visibility should be typographically unambiguous	MISRA C:2012 Dir 4.5
MISRA C:2012 Dir 4.6	typedefs that indicate size and signedness should be used in place of the basic numerical types	MISRA C:2012 Dir 4.6
MISRA C:2012 Dir 4.7	If a function returns error information, then that error information shall be tested	MISRA C:2012 Dir 4.7
MISRA C:2012 Dir 4.8	If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden	MISRA C:2012 Dir 4.8
MISRA C:2012 Dir 4.9	A function should be used in preference to a function-like macro where they are interchangeable	MISRA C:2012 Dir 4.9

See Also

Check MISRA C:2012 (-misra3)

More About

- "Check for and Review Coding Standard Violations" on page 16-2
- "Coding Standards"
- "Decidable MISRA C:2012 Rules Supported by Polyspace Bug Finder" on page 17-54
- "Required and Statically Enforceable CERT C Rules Supported by Polyspace Bug Finder" on page 17-75
- "Checkers Deactivated in Polyspace as You Code Analysis" on page 11-78

Polyspace Support for MISRA C: 2012 Amendments

Starting in R2021a, Polyspace Bug Finder supports amendments to MISRA C:2012 rules in Technical Corrigendum 1 and Amendments 1 and 2.

MISRA C:2012 Technical Corrigendum 1

MISRA C:2012 Technical Corrigendum 1 adds clarifications to existing rules. The clarifications have led to corresponding changes in the Polyspace checkers.

Rule	Description	Update in Technical Corrigendum 1
MISRA C:2012 Rule 10.1	Operands shall not be of an inappropriate essential type.	The rule now explicitly forbids use of pointer types with logical operands such as <code>&&</code> , <code> </code> and <code>!</code> .
MISRA C:2012 Rule 10.5	The value of an expression should not be cast to an inappropriate essential type.	The rule now forbids casts of integer constants with value 0 or 1 to essentially enum types.
MISRA C:2012 Rule 11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type.	The rule now takes into account only the unqualified types that the pointers point to. For instance, if a pointer is assigned to another and the only difference between the pointed types is a <code>const</code> qualifier, the rule does not consider this assignment as a conversion.
MISRA C:2012 Rule 11.4	A conversion should not be performed between a pointer to object and an integer type.	The rule now applies explicitly to pointers to objects only. Conversions between an integer type and other pointer types such as <code>void*</code> or pointers to functions are flagged by other rules.
MISRA C:2012 Rule 11.9	The macro <code>NULL</code> shall be the only permitted form of integer null pointer constant.	The rule allows the use of <code>{0}</code> to initialize aggregates or unions containing pointers.
MISRA C:2012 Rule 14.2	A for loop shall be well-formed.	The rule allows any form of initialization of the loop counter as long as the initialization does not have other side effects.

MISRA C: 2012 Amendment 1 (AMD1)

MISRA C:2012 Amendment 1 introduces new coding rules that prevent security vulnerabilities.

Rule	Description
MISRA C:2012 Dir 4.14	The validity of values received from external sources shall be checked.
MISRA C:2012 Rule 12.5	The <code>sizeof</code> operator shall not have an operand which is a function parameter declared as "array of type".
MISRA C:2012 Rule 21.13	Any value passed to a function in <code><ctype.h></code> shall be representable as an <code>unsigned char</code> or be the value EOF.
MISRA C:2012 Rule 21.14	The Standard Library function <code>memcmp</code> shall not be used to compare null terminated strings.
MISRA C:2012 Rule 21.15	The pointer arguments to the Standard Library functions <code>memcpy</code> , <code>memmove</code> and <code>memcmp</code> shall be pointers to qualified or unqualified versions of compatible types.
MISRA C:2012 Rule 21.16	The pointer arguments to the Standard Library function <code>memcmp</code> shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type.
MISRA C:2012 Rule 21.17	Use of the string handling function from <code><string.h></code> shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters.
MISRA C:2012 Rule 21.18	The <code>size_t</code> argument passed to any function in <code><string.h></code> shall have an appropriate value.
MISRA C:2012 Rule 21.19	The pointers returned by the Standard Library functions <code>localeconv</code> , <code>getenv</code> , <code>setlocale</code> or <code>strerror</code> shall only be used as if they have pointer to <code>const</code> -qualified type.
MISRA C:2012 Rule 21.20	The pointer returned by the Standard Library functions <code>asctime</code> , <code>ctime</code> , <code>gmtime</code> , <code>localtime</code> , <code>localeconv</code> , <code>getenv</code> , <code>setlocale</code> or <code>strerror</code> shall not be used following a subsequent call to the same function.
MISRA C:2012 Rule 22.7	The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF.
MISRA C:2012 Rule 22.8	The value of <code>errno</code> shall be set to zero prior to a call to an <code>errno</code> -setting-function.
MISRA C:2012 Rule 22.9	The value of <code>errno</code> shall be tested against zero after calling an <code>errno</code> -setting function.
MISRA C:2012 Rule 22.10	The value of <code>errno</code> shall only be tested when the last function to be called was an <code>errno</code> -setting function.

MISRA C:2012 Amendment 1 introduced changes to existing rules:

Rule	Description	Change
MISRA C:2012 Rule 21.8	The Standard Library termination functions of <code><stdlib.h></code> shall not be used	References to <code>getenv</code> is removed from rule headline and amplification. After the introduction of 21.19 and 21.20, using <code>getenv</code> in 21.8 is no longer necessary.

MISRA C:2012 Amendment 2 (AMD2)

MISRA C:2012 Amendment 2 addresses the new language features in the C11 standard. All updates in Amendment 2 have been incorporated in the Polyspace checkers.

Rule	Description	Update in Amendment 2
MISRA C:2012 Rule 1.4	Emergent language features shall not be used.	This rule is new in Amendment 2.
MISRA C:2012 Rule 12.1	The precedence of operators within expressions should be made explicit.	The rule now mandates a violation if the operand of the <code>_Alignof</code> operator is not enclosed in parenthesis.
MISRA C:2012 Rule 21.3	The memory allocation and deallocation functions of <code><stdlib.h></code> shall not be used.	The rule now flags uses of the <code>aligned_alloc</code> function.
MISRA C:2012 Rule 21.8	The Standard Library termination functions of <code><stdlib.h></code> shall not be used.	The rule no longer flags <code>system</code> . In addition to <code>exit</code> and <code>abort</code> , the rule now flags <code>_Exit</code> and <code>quick_exit</code> .
MISRA C:2012 Rule 21.21	The Standard Library function <code>system</code> of <code><stdlib.h></code> shall not be used.	This rule is new in Amendment 2.
MISRA C:2012 Rule 22.1	All resources obtained dynamically by means of Standard Library functions shall be explicitly released.	The rule now flags memory allocation using the <code>aligned_alloc</code> function if the memory is not released.

Essential Types in MISRA C:2012 Rules 10.x

MISRA C:2012 rules 10.x classify data types in categories. The rules treat data types in the same category as essentially similar.

For instance, the data types `float`, `double` and `long double` are considered as essentially floating. Rule 10.1 states that the `%` operation must not have essentially floating operands. This statement implies that the operands cannot have one of these three data types: `float`, `double` and `long double`.

Categories of Essential Types

The essential types fall in these categories:

Essential type category	Standard types
Essentially Boolean	<code>bool</code> or <code>_Bool</code> (defined in <code>stdbool.h</code>) If you define a boolean type through a <code>typedef</code> , you must specify this type name before coding rules checking. For more information, see Effective boolean types (-boolean-types) ..
Essentially character	<code>char</code>
Essentially enum	named enum
Essentially signed	<code>signed char</code> , <code>signed short</code> , <code>signed int</code> , <code>signed long</code> , <code>signed long long</code>
Essentially unsigned	<code>unsigned char</code> , <code>unsigned short</code> , <code>unsigned int</code> , <code>unsigned long</code> , <code>unsigned long long</code>
Essentially floating	<code>float</code> , <code>double</code> , <code>long double</code>

How MISRA C:2012 Uses Essential Types

These rules use essential types in their statements:

- MISRA C:2012 Rule 10.1: Operands shall not be of an inappropriate essential type.

For instance, the right operand of the `<<` or `>>` operator must be essentially unsigned. Otherwise, negative values can cause undefined behavior.

- MISRA C:2012 Rule 10.2: Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.

For instance, the type `char` does not represent numeric values. Do not use a variable of this type in addition and subtraction operations.

- MISRA C:2012 Rule 10.3: The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.

For instance, do not assign a variable of data type `double` to a variable with the narrower data type `float`.

- MISRA C:2012 Rule 10.4: Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.

For instance, do not perform an addition operation with a signed `int` operand, which belongs to the essentially signed category, and an unsigned `int` operand, which belongs to the essentially unsigned category.

- MISRA C:2012 Rule 10.5: The value of an expression should not be cast to an inappropriate essential type.

For instance, do not perform a cast between essentially floating types and essentially character types.

- MISRA C:2012 Rule 10.6: The value of a composite expression shall not be assigned to an object with wider essential type.

For instance, if a multiplication, binary addition or bitwise operation involves unsigned `char` operands, do not assign the result to a variable having the wider type unsigned `int`.

- MISRA C:2012 Rule 10.7: If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.

For instance, if one operand of an addition operation is a composite expression with two unsigned `char` operands, the other operand must not have the wider type unsigned `int`.

See Also

More About

- “Check for Coding Rule Violations” on page 6-22
- “MISRA C:2012 Directives and Rules”

Unsupported MISRA C:2012 Guidelines

The Polyspace coding rules checkers do not check the following MISRA C:2012 directives. These directives cannot be enforced because they are outside the scope of Polyspace software. These guidelines concern documentation, dynamic aspects, or functional aspects of MISRA rules.

For the list of supported rules and directives, see “MISRA C:2012 Directives and Rules”.

Number	Category	AGC Category	Definition
Directive 3.1	Required	Required	All code shall be traceable to documented requirements
Directive 4.2	Advisory	Advisory	All usage of assembly language should be documented

See Also

More About

- “MISRA C:2012 Directives and Rules”

Required and Statically Enforceable CERT C Rules Supported by Polyspace Bug Finder

The CERT C standard classifies the guidelines that compliant C code must follow as **Rules**. These **Rules** are also considered enforceable by static analysis. Polyspace supports 120 out of 120 such guidelines.

CERT C Rule	Description	Polyspace Checker
CERT C: Rule ARR30-C	Do not form or use out-of-bounds pointers or array subscripts	CERT C: Rule ARR30-C
CERT C: Rule ARR32-C	Ensure size arguments for variable length arrays are in a valid range	CERT C: Rule ARR32-C
CERT C: Rule ARR36-C	Do not subtract or compare two pointers that do not refer to the same array	CERT C: Rule ARR36-C
CERT C: Rule ARR37-C	Do not add or subtract an integer to a pointer to a non-array object	CERT C: Rule ARR37-C
CERT C: Rule ARR38-C	Guarantee that library functions do not form invalid pointers	CERT C: Rule ARR38-C
CERT C: Rule ARR39-C	Do not add or subtract a scaled integer to a pointer	CERT C: Rule ARR39-C
CERT C: Rule CON30-C	Clean up thread-specific storage	CERT C: Rule CON30-C
CERT C: Rule CON31-C	Do not destroy a mutex while it is locked	CERT C: Rule CON31-C
CERT C: Rule CON32-C	Prevent data races when accessing bit fields from multiple threads	CERT C: Rule CON32-C
CERT C: Rule CON33-C	Avoid race conditions when using library functions	CERT C: Rule CON33-C
CERT C: Rule CON34-C	Declare objects shared between threads with appropriate storage durations	CERT C: Rule CON34-C
CERT C: Rule CON35-C	Avoid deadlock by locking in a predefined order	CERT C: Rule CON35-C
CERT C: Rule CON36-C	Wrap functions that can spuriously wake up in a loop	CERT C: Rule CON36-C
CERT C: Rule CON37-C	Do not call signal() in a multithreaded program	CERT C: Rule CON37-C
CERT C: Rule CON38-C	Preserve thread safety and liveness when using condition variables	CERT C: Rule CON38-C

CERT C Rule	Description	Polyspace Checker
CERT C: Rule CON39-C	Do not join or detach a thread that was previously joined or detached	CERT C: Rule CON39-C
CERT C: Rule CON40-C	Do not refer to an atomic variable twice in an expression	CERT C: Rule CON40-C
CERT C: Rule CON41-C	Wrap functions that can fail spuriously in a loop	CERT C: Rule CON41-C
CERT C: Rule CON43-C	Do not allow data races in multithreaded code	CERT C: Rule CON43-C
CERT C: Rule DCL30-C	Declare objects with appropriate storage durations	CERT C: Rule DCL30-C
CERT C: Rule DCL31-C	Declare identifiers before using them	CERT C: Rule DCL31-C
CERT C: Rule DCL36-C	Do not declare an identifier with conflicting linkage classifications	CERT C: Rule DCL36-C
CERT C: Rule DCL37-C	Do not declare or define a reserved identifier	CERT C: Rule DCL37-C
CERT C: Rule DCL38-C	Use the correct syntax when declaring a flexible array member	CERT C: Rule DCL38-C
CERT C: Rule DCL39-C	Avoid information leakage in structure padding	CERT C: Rule DCL39-C
CERT C: Rule DCL40-C	Do not create incompatible declarations of the same function or object	CERT C: Rule DCL40-C
CERT C: Rule DCL41-C	Do not declare variables inside a switch statement before the first case label	CERT C: Rule DCL41-C
CERT C: Rule ENV30-C	Do not modify the object referenced by the return value of certain functions	CERT C: Rule ENV30-C
CERT C: Rule ENV31-C	Do not rely on an environment pointer following an operation that may invalidate it	CERT C: Rule ENV31-C
CERT C: Rule ENV32-C	All exit handlers must return normally	CERT C: Rule ENV32-C
CERT C: Rule ENV33-C	Do not call system()	CERT C: Rule ENV33-C
CERT C: Rule ENV34-C	Do not store pointers returned by certain functions	CERT C: Rule ENV34-C

CERT C Rule	Description	Polyspace Checker
CERT C: Rule ERR30-C	Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure	CERT C: Rule ERR30-C
CERT C: Rule ERR32-C	Do not rely on indeterminate values of errno	CERT C: Rule ERR32-C
CERT C: Rule ERR33-C	Detect and handle standard library errors	CERT C: Rule ERR33-C
CERT C: Rule ERR34-C	Detect errors when converting a string to a number	CERT C: Rule ERR34-C
CERT C: Rule EXP30-C	Do not depend on the order of evaluation for side effects	CERT C: Rule EXP30-C
CERT C: Rule EXP32-C	Do not access a volatile object through a nonvolatile reference	CERT C: Rule EXP32-C
CERT C: Rule EXP33-C	Do not read uninitialized memory	CERT C: Rule EXP33-C
CERT C: Rule EXP34-C	Do not dereference null pointers	CERT C: Rule EXP34-C
CERT C: Rule EXP35-C	Do not modify objects with temporary lifetime	CERT C: Rule EXP35-C
CERT C: Rule EXP36-C	Do not cast pointers into more strictly aligned pointer types	CERT C: Rule EXP36-C
CERT C: Rule EXP37-C	Call functions with the correct number and type of arguments	CERT C: Rule EXP37-C
CERT C: Rule EXP39-C	Do not access a variable through a pointer of an incompatible type	CERT C: Rule EXP39-C
CERT C: Rule EXP40-C	Do not modify constant objects	CERT C: Rule EXP40-C
CERT C: Rule EXP42-C	Do not compare padding data	CERT C: Rule EXP42-C
CERT C: Rule EXP43-C	Avoid undefined behavior when using restrict-qualified pointers	CERT C: Rule EXP43-C
CERT C: Rule EXP44-C	Do not rely on side effects in operands to sizeof, _Alignof, or _Generic	CERT C: Rule EXP44-C
CERT C: Rule EXP45-C	Do not perform assignments in selection statements	CERT C: Rule EXP45-C
CERT C: Rule EXP46-C	Do not use a bitwise operator with a Boolean-like operand	CERT C: Rule EXP46-C
CERT C: Rule EXP47-C	Do not call va_arg with an argument of the incorrect type	CERT C: Rule EXP47-C
CERT C: Rule FIO30-C	Exclude user input from format strings	CERT C: Rule FIO30-C

CERT C Rule	Description	Polyspace Checker
CERT C: Rule FIO32-C	Do not perform operations on devices that are only appropriate for files	CERT C: Rule FI032-C
CERT C: Rule FIO34-C	Distinguish between characters read from a file and EOF or WEOF	CERT C: Rule FI034-C
CERT C: Rule FIO37-C	Do not assume that fgets() or fgetws() returns a nonempty string when successful	CERT C: Rule FI037-C
CERT C: Rule FIO38-C	Do not copy a FILE object	CERT C: Rule FI038-C
CERT C: Rule FIO39-C	Do not alternately input and output from a stream without an intervening flush or positioning call	CERT C: Rule FI039-C
CERT C: Rule FIO40-C	Reset strings on fgets() or fgetws() failure	CERT C: Rule FI040-C
CERT C: Rule FIO41-C	Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects	CERT C: Rule FI041-C
CERT C: Rule FIO42-C	Close files when they are no longer needed	CERT C: Rule FI042-C
CERT C: Rule FIO44-C	Only use values for fsetpos() that are returned from fgetpos()	CERT C: Rule FI044-C
CERT C: Rule FIO45-C	Avoid TOCTOU race conditions while accessing files	CERT C: Rule FI045-C
CERT C: Rule FIO46-C	Do not access a closed file	CERT C: Rule FI046-C
CERT C: Rule FIO47-C	Use valid format strings	CERT C: Rule FI047-C
CERT C: Rule FLP30-C	Do not use floating-point variables as loop counters	CERT C: Rule FLP30-C
CERT C: Rule FLP32-C	Prevent or detect domain and range errors in math functions	CERT C: Rule FLP32-C
CERT C: Rule FLP34-C	Ensure that floating-point conversions are within range of the new type	CERT C: Rule FLP34-C
CERT C: Rule FLP36-C	Preserve precision when converting integral values to floating-point type	CERT C: Rule FLP36-C
CERT C: Rule FLP37-C	Do not use object representations to compare floating-point values	CERT C: Rule FLP37-C
CERT C: Rule INT30-C	Ensure that unsigned integer operations do not wrap	CERT C: Rule INT30-C

CERT C Rule	Description	Polyspace Checker
CERT C: Rule INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data	CERT C: Rule INT31-C
CERT C: Rule INT32-C	Ensure that operations on signed integers do not result in overflow	CERT C: Rule INT32-C
CERT C: Rule INT33-C	Ensure that division and remainder operations do not result in divide-by-zero errors	CERT C: Rule INT33-C
CERT C: Rule INT34-C	Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand	CERT C: Rule INT34-C
CERT C: Rule INT35-C	Use correct integer precisions	CERT C: Rule INT35-C
CERT C: Rule INT36-C	Converting a pointer to integer or integer to pointer	CERT C: Rule INT36-C
CERT C: Rule MEM30-C	Do not access freed memory	CERT C: Rule MEM30-C
CERT C: Rule MEM31-C	Free dynamically allocated memory when no longer needed	CERT C: Rule MEM31-C
CERT C: Rule MEM33-C	Allocate and copy structures containing a flexible array member dynamically	CERT C: Rule MEM33-C
CERT C: Rule MEM34-C	Only free memory allocated dynamically	CERT C: Rule MEM34-C
CERT C: Rule MEM35-C	Allocate sufficient memory for an object	CERT C: Rule MEM35-C
CERT C: Rule MEM36-C	Do not modify the alignment of objects by calling realloc()	CERT C: Rule MEM36-C
CERT C: Rule MSC30-C	Do not use the rand() function for generating pseudorandom numbers	CERT C: Rule MSC30-C
CERT C: Rule MSC32-C	Properly seed pseudorandom number generators	CERT C: Rule MSC32-C
CERT C: Rule MSC33-C	Do not pass invalid data to the asctime() function	CERT C: Rule MSC33-C
CERT C: Rule MSC37-C	Ensure that control never reaches the end of a non-void function	CERT C: Rule MSC37-C
CERT C: Rule MSC38-C	Do not treat a predefined identifier as an object if it might only be implemented as a macro	CERT C: Rule MSC38-C
CERT C: Rule MSC39-C	Do not call va_arg() on a va_list that has an indeterminate value	CERT C: Rule MSC39-C

CERT C Rule	Description	Polyspace Checker
CERT C: Rule MSC40-C	Do not violate constraints	CERT C: Rule MSC40-C
CERT C: Rule MSC41-C	Never hard code sensitive information	CERT C: Rule MSC41-C
CERT C: Rule POS30-C	Use the readlink() function properly	CERT C: Rule POS30-C
CERT C: Rule POS34-C	Do not call putenv() with a pointer to an automatic variable as the argument	CERT C: Rule POS34-C
CERT C: Rule POS35-C	Avoid race conditions while checking for the existence of a symbolic link	CERT C: Rule POS35-C
CERT C: Rule POS36-C	Observe correct revocation order while relinquishing privileges	CERT C: Rule POS36-C
CERT C: Rule POS37-C	Ensure that privilege relinquishment is successful	CERT C: Rule POS37-C
CERT C: Rule POS38-C	Beware of race conditions when using fork and file descriptors	CERT C: Rule POS38-C
CERT C: Rule POS39-C	Use the correct byte ordering when transferring data between systems	CERT C: Rule POS39-C
CERT C: Rule POS44-C	Do not use signals to terminate threads	CERT C: Rule POS44-C
CERT C: Rule POS47-C	Do not use threads that can be canceled asynchronously	CERT C: Rule POS47-C
CERT C: Rule POS48-C	Do not unlock or destroy another POSIX thread's mutex	CERT C: Rule POS48-C
CERT C: Rule POS49-C	When data must be accessed by multiple threads, provide a mutex and guarantee no adjacent data is also accessed	CERT C: Rule POS49-C
CERT C: Rule POS50-C	Declare objects shared between POSIX threads with appropriate storage durations	CERT C: Rule POS50-C
CERT C: Rule POS51-C	Avoid deadlock with POSIX threads by locking in predefined order	CERT C: Rule POS51-C
CERT C: Rule POS52-C	Do not perform operations that can block while holding a POSIX lock	CERT C: Rule POS52-C
CERT C: Rule POS53-C	Do not use more than one mutex for concurrent waiting operations on a condition variable	CERT C: Rule POS53-C

CERT C Rule	Description	Polyspace Checker
CERT C: Rule POS54-C	Detect and handle POSIX library errors	CERT C: Rule POS54-C
CERT C: Rule PRE30-C	Do not create a universal character name through concatenation	CERT C: Rule PRE30-C
CERT C: Rule PRE31-C	Avoid side effects in arguments to unsafe macros	CERT C: Rule PRE31-C
CERT C: Rule PRE32-C	Do not use preprocessor directives in invocations of function-like macros	CERT C: Rule PRE32-C
CERT C: Rule SIG30-C	Call only asynchronous-safe functions within signal handlers	CERT C: Rule SIG30-C
CERT C: Rule SIG31-C	Do not access shared objects in signal handlers	CERT C: Rule SIG31-C
CERT C: Rule SIG34-C	Do not call signal() from within interruptible signal handlers	CERT C: Rule SIG34-C
CERT C: Rule SIG35-C	Do not return from a computational exception signal handler	CERT C: Rule SIG35-C
CERT C: Rule STR30-C	Do not attempt to modify string literals	CERT C: Rule STR30-C
CERT C: Rule STR31-C	Guarantee that storage for strings has sufficient space for character data and the null terminator	CERT C: Rule STR31-C
CERT C: Rule STR32-C	Do not pass a non-null-terminated character sequence to a library function that expects a string	CERT C: Rule STR32-C
CERT C: Rule STR34-C	Cast characters to unsigned char before converting to larger integer sizes	CERT C: Rule STR34-C
CERT C: Rule STR37-C	Arguments to character-handling functions must be representable as an unsigned char	CERT C: Rule STR37-C
CERT C: Rule STR38-C	Do not confuse narrow and wide character strings and functions	CERT C: Rule STR38-C
CERT C: Rule WIN30-C	Properly pair allocation and deallocation functions	CERT C: Rule WIN30-C

See Also

Check SEI CERT-C (-cert-c)

More About

- “Check for and Review Coding Standard Violations” on page 16-2
- “Coding Standards”
- “Required or Mandatory MISRA C:2012 Rules Supported by Polyspace Bug Finder” on page 17-43
- “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78

Required MISRA C++:2008 Coding Rules Supported by Polyspace Bug Finder

The MISRA C++:2008 standard classifies the rules that compliant C++ code must follow as **Required**. Polyspace Bug Finder supports 195 out of 198 required MISRA C++:2008 coding rules.

Supported Rules

Polyspace supports these **Required** rules.

MISRA C++:2008 Rule	Description	Polyspace Checker		
MISRA C++:2008 Rule 0-1-1	A project shall not contain unreachable code	MISRA 0-1-1	C++:2008	Rule
MISRA C++:2008 Rule 0-1-10	Every defined function shall be called at least once	MISRA 0-1-10	C++:2008	Rule
MISRA C++:2008 Rule 0-1-11	There shall be no unused parameters (named or unnamed) in nonvirtual functions	MISRA 0-1-11	C++:2008	Rule
MISRA C++:2008 Rule 0-1-12	There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it	MISRA 0-1-12	C++:2008	Rule
MISRA C++:2008 Rule 0-1-2	A project shall not contain infeasible paths	MISRA 0-1-2	C++:2008	Rule
MISRA C++:2008 Rule 0-1-3	A project shall not contain unused variables	MISRA 0-1-3	C++:2008	Rule
MISRA C++:2008 Rule 0-1-4	A project shall not contain non-volatile POD variables having only one use	MISRA 0-1-4	C++:2008	Rule
MISRA C++:2008 Rule 0-1-5	A project shall not contain unused type declarations	MISRA 0-1-5	C++:2008	Rule
MISRA C++:2008 Rule 0-1-6	A project shall not contain instances of non-volatile variables being given values that are never subsequently used.	MISRA 0-1-6	C++:2008	Rule
MISRA C++:2008 Rule 0-1-7	The value returned by a function having a non-void return type that is not an overloaded operator shall always be used	MISRA 0-1-7	C++:2008	Rule
MISRA C++:2008 Rule 0-1-8	All functions with void return type shall have external side effect(s)	MISRA 0-1-8	C++:2008	Rule

MISRA C++:2008 Rule	Description	Polyspace Checker		
MISRA C++:2008 Rule 0-1-9	There shall be no dead code	MISRA 0-1-9	C++:2008	Rule
MISRA C++:2008 Rule 0-2-1	An object shall not be assigned to an overlapping object	MISRA 0-2-1	C++:2008	Rule
MISRA C++:2008 Rule 0-3-2	If a function generates error information, then that error information shall be tested	MISRA 0-3-2	C++:2008	Rule
MISRA C++:2008 Rule 10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy	MISRA 10-1-2	C++:2008	Rule
MISRA C++:2008 Rule 10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy	MISRA 10-1-3	C++:2008	Rule
MISRA C++:2008 Rule 10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy	MISRA 10-3-1	C++:2008	Rule
MISRA C++:2008 Rule 10-3-2	Each overriding virtual function shall be declared with the virtual keyword	MISRA 10-3-2	C++:2008	Rule
MISRA C++:2008 Rule 10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual	MISRA 10-3-3	C++:2008	Rule
MISRA C++:2008 Rule 11-0-1	Member data in non- POD class types shall be private	MISRA 11-0-1	C++:2008	Rule
MISRA C++:2008 Rule 12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor	MISRA 12-1-1	C++:2008	Rule
MISRA C++:2008 Rule 12-1-3	All constructors that are callable with a single argument of fundamental type shall be declared explicit	MISRA 12-1-3	C++:2008	Rule
MISRA C++:2008 Rule 12-8-1	A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member	MISRA 12-8-1	C++:2008	Rule
MISRA C++:2008 Rule 12-8-2	The copy assignment operator shall be declared protected or private in an abstract class	MISRA 12-8-2	C++:2008	Rule
MISRA C++:2008 Rule 14-5-1	A non-member generic function shall only be declared in a namespace that is not an associated namespace	MISRA 14-5-1	C++:2008	Rule

MISRA C++:2008 Rule	Description	Polyspace Checker		
MISRA C++:2008 Rule 14-5-2	A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter	MISRA 14-5-2	C++:2008	Rule
MISRA C++:2008 Rule 14-5-3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter	MISRA 14-5-3	C++:2008	Rule
MISRA C++:2008 Rule 14-6-1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->	MISRA 14-6-1	C++:2008	Rule
MISRA C++:2008 Rule 14-6-2	The function chosen by overload resolution shall resolve to a function declared previously in the translation unit	MISRA 14-6-2	C++:2008	Rule
MISRA C++:2008 Rule 14-7-3	All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template	MISRA 14-7-3	C++:2008	Rule
MISRA C++:2008 Rule 14-8-1	Overloaded function templates shall not be explicitly specialized	MISRA 14-8-1	C++:2008	Rule
MISRA C++:2008 Rule 15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement	MISRA 15-0-3	C++:2008	Rule
MISRA C++:2008 Rule 15-1-1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown	MISRA 15-1-1	C++:2008	Rule
MISRA C++:2008 Rule 15-1-2	NULL shall not be thrown explicitly	MISRA 15-1-2	C++:2008	Rule
MISRA C++:2008 Rule 15-1-3	An empty throw (throw;) shall only be used in the compound-statement of a catch handler	MISRA 15-1-3	C++:2008	Rule
MISRA C++:2008 Rule 15-3-1	Exceptions shall be raised only after start-up and before termination of the program	MISRA 15-3-1	C++:2008	Rule

MISRA C++:2008 Rule	Description	Polyspace Checker		
MISRA C++:2008 Rule 15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases	MISRA 15-3-3	C++:2008	Rule
MISRA C++:2008 Rule 15-3-4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point	MISRA 15-3-4	C++:2008	Rule
MISRA C++:2008 Rule 15-3-5	A class type exception shall always be caught by reference	MISRA 15-3-5	C++:2008	Rule
MISRA C++:2008 Rule 15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class	MISRA 15-3-6	C++:2008	Rule
MISRA C++:2008 Rule 15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last	MISRA 15-3-7	C++:2008	Rule
MISRA C++:2008 Rule 15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids	MISRA 15-4-1	C++:2008	Rule
MISRA C++:2008 Rule 15-5-1	A class destructor shall not exit with an exception	MISRA 15-5-1	C++:2008	Rule
MISRA C++:2008 Rule 15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s)	MISRA 15-5-2	C++:2008	Rule
MISRA C++:2008 Rule 15-5-3	The terminate() function shall not be called implicitly	MISRA 15-5-3	C++:2008	Rule
MISRA C++:2008 Rule 16-0-1	#include directives in a file shall only be preceded by other preprocessor directives or comments	MISRA 16-0-1	C++:2008	Rule

MISRA C++:2008 Rule	Description	Polyspace Checker		
MISRA C++:2008 Rule 16-0-2	Macros shall only be #define 'd or #undef 'd in the global namespace	MISRA 16-0-2	C++:2008	Rule
MISRA C++:2008 Rule 16-0-3	#undef shall not be used	MISRA 16-0-3	C++:2008	Rule
MISRA C++:2008 Rule 16-0-4	Function-like macros shall not be defined	MISRA 16-0-4	C++:2008	Rule
MISRA C++:2008 Rule 16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives	MISRA 16-0-5	C++:2008	Rule
MISRA C++:2008 Rule 16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##	MISRA 16-0-6	C++:2008	Rule
MISRA C++:2008 Rule 16-0-7	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator	MISRA 16-0-7	C++:2008	Rule
MISRA C++:2008 Rule 16-0-8	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token	MISRA 16-0-8	C++:2008	Rule
MISRA C++:2008 Rule 16-1-1	The defined preprocessor operator shall only be used in one of the two standard forms	MISRA 16-1-1	C++:2008	Rule
MISRA C++:2008 Rule 16-1-2	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related	MISRA 16-1-2	C++:2008	Rule
MISRA C++:2008 Rule 16-2-1	The preprocessor shall only be used for file inclusion and include guards	MISRA 16-2-1	C++:2008	Rule
MISRA C++:2008 Rule 16-2-2	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers	MISRA 16-2-2	C++:2008	Rule
MISRA C++:2008 Rule 16-2-3	Include guards shall be provided	MISRA 16-2-3	C++:2008	Rule
MISRA C++:2008 Rule 16-2-4	The ', ", /* or // characters shall not occur in a header file name	MISRA 16-2-4	C++:2008	Rule

MISRA C++:2008 Rule	Description	Polyspace Checker		
MISRA C++:2008 Rule 16-2-6	The #include directive shall be followed by either a <filename> or "filename" sequence	MISRA 16-2-6	C++:2008	Rule
MISRA C++:2008 Rule 16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition	MISRA 16-3-1	C++:2008	Rule
MISRA C++:2008 Rule 17-0-1	Reserved identifiers, macros and functions in the Standard Library shall not be defined, redefined or undefined	MISRA 17-0-1	C++:2008	Rule
MISRA C++:2008 Rule 17-0-2	The names of standard library macros and objects shall not be reused	MISRA 17-0-2	C++:2008	Rule
MISRA C++:2008 Rule 17-0-3	The names of standard library functions shall not be overridden	MISRA 17-0-3	C++:2008	Rule
MISRA C++:2008 Rule 17-0-5	The setjmp macro and the longjmp function shall not be used	MISRA 17-0-5	C++:2008	Rule
MISRA C++:2008 Rule 18-0-1	The C library shall not be used	MISRA 18-0-1	C++:2008	Rule
MISRA C++:2008 Rule 18-0-2	The library functions atof, atoi and atol from library <cstdlib> shall not be used	MISRA 18-0-2	C++:2008	Rule
MISRA C++:2008 Rule 18-0-3	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used	MISRA 18-0-3	C++:2008	Rule
MISRA C++:2008 Rule 18-0-4	The time handling functions of library <ctime> shall not be used	MISRA 18-0-4	C++:2008	Rule
MISRA C++:2008 Rule 18-0-5	The unbounded functions of library <cstring> shall not be used	MISRA 18-0-5	C++:2008	Rule
MISRA C++:2008 Rule 18-2-1	The macro offsetof shall not be used	MISRA 18-2-1	C++:2008	Rule
MISRA C++:2008 Rule 18-4-1	Dynamic heap memory allocation shall not be used	MISRA 18-4-1	C++:2008	Rule
MISRA C++:2008 Rule 18-7-1	The signal handling facilities of <csignal> shall not be used	MISRA 18-7-1	C++:2008	Rule
MISRA C++:2008 Rule 19-3-1	The error indicator errno shall not be used	MISRA 19-3-1	C++:2008	Rule

MISRA C++:2008 Rule	Description	Polyspace Checker		
MISRA C++:2008 Rule 1-0-1	All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1"	MISRA 1-0-1	C++:2008	Rule
MISRA C++:2008 Rule 27-0-1	The stream input/output library <cstdio> shall not be used	MISRA 27-0-1	C++:2008	Rule
MISRA C++:2008 Rule 2-10-1	Different identifiers shall be typographically unambiguous	MISRA 2-10-1	C++:2008	Rule
MISRA C++:2008 Rule 2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope	MISRA 2-10-2	C++:2008	Rule
MISRA C++:2008 Rule 2-10-3	A typedef name (including qualification, if any) shall be a unique identifier	MISRA 2-10-3	C++:2008	Rule
MISRA C++:2008 Rule 2-10-4	A class, union or enum name (including qualification, if any) shall be a unique identifier	MISRA 2-10-4	C++:2008	Rule
MISRA C++:2008 Rule 2-10-6	If an identifier refers to a type, it shall not also refer to an object or a function in the same scope	MISRA 2-10-6	C++:2008	Rule
MISRA C++:2008 Rule 2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used	MISRA 2-13-1	C++:2008	Rule
MISRA C++:2008 Rule 2-13-2	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used	MISRA 2-13-2	C++:2008	Rule
MISRA C++:2008 Rule 2-13-3	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type	MISRA 2-13-3	C++:2008	Rule
MISRA C++:2008 Rule 2-13-4	Literal suffixes shall be upper case	MISRA 2-13-4	C++:2008	Rule
MISRA C++:2008 Rule 2-13-5	Narrow and wide string literals shall not be concatenated	MISRA 2-13-5	C++:2008	Rule
MISRA C++:2008 Rule 2-3-1	Trigraphs shall not be used	MISRA 2-3-1	C++:2008	Rule
MISRA C++:2008 Rule 2-7-1	The character sequence /* shall not be used within a C-style comment	MISRA 2-7-1	C++:2008	Rule
MISRA C++:2008 Rule 2-7-2	Sections of code shall not be "commented out" using C-style comments	MISRA 2-7-2	C++:2008	Rule

MISRA C++:2008 Rule	Description	Polyspace Checker		
MISRA C++:2008 Rule 3-1-1	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule	MISRA 3-1-1	C++:2008	Rule
MISRA C++:2008 Rule 3-1-2	Functions shall not be declared at block scope	MISRA 3-1-2	C++:2008	Rule
MISRA C++:2008 Rule 3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization	MISRA 3-1-3	C++:2008	Rule
MISRA C++:2008 Rule 3-2-1	All declarations of an object or function shall have compatible types	MISRA 3-2-1	C++:2008	Rule
MISRA C++:2008 Rule 3-2-2	The One Definition Rule shall not be violated	MISRA 3-2-2	C++:2008	Rule
MISRA C++:2008 Rule 3-2-3	A type, object or function that is used in multiple translation units shall be declared in one and only one file	MISRA 3-2-3	C++:2008	Rule
MISRA C++:2008 Rule 3-2-4	An identifier with external linkage shall have exactly one definition	MISRA 3-2-4	C++:2008	Rule
MISRA C++:2008 Rule 3-3-1	Objects or functions with external linkage shall be declared in a header file	MISRA 3-3-1	C++:2008	Rule
MISRA C++:2008 Rule 3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier	MISRA 3-3-2	C++:2008	Rule
MISRA C++:2008 Rule 3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility	MISRA 3-4-1	C++:2008	Rule
MISRA C++:2008 Rule 3-9-1	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations	MISRA 3-9-1	C++:2008	Rule
MISRA C++:2008 Rule 3-9-3	The underlying bit representations of floating-point values shall not be used	MISRA 3-9-3	C++:2008	Rule
MISRA C++:2008 Rule 4-10-1	NULL shall not be used as an integer value	MISRA 4-10-1	C++:2008	Rule

MISRA C++:2008 Rule	Description	Polyspace Checker		
MISRA C++:2008 Rule 4-10-2	Literal zero (0) shall not be used as the null-pointer-constant	MISRA 4-10-2	C++:2008	Rule
MISRA C++:2008 Rule 4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, , !, the equality operators == and !=, the unary & operator, and the conditional operator	MISRA 4-5-1	C++:2008	Rule
MISRA C++:2008 Rule 4-5-2	Expressions with type enum shall not be used as operands to built-in operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=	MISRA 4-5-2	C++:2008	Rule
MISRA C++:2008 Rule 4-5-3	Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator N	MISRA 4-5-3	C++:2008	Rule
MISRA C++:2008 Rule 5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits	MISRA 5-0-1	C++:2008	Rule
MISRA C++:2008 Rule 5-0-10	If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand	MISRA 5-0-10	C++:2008	Rule
MISRA C++:2008 Rule 5-0-11	The plain char type shall only be used for the storage and use of character values	MISRA 5-0-11	C++:2008	Rule
MISRA C++:2008 Rule 5-0-12	Signed char and unsigned char type shall only be used for the storage and use of numeric values	MISRA 5-0-12	C++:2008	Rule

MISRA C++:2008 Rule	Description	Polyspace Checker		
MISRA C++:2008 Rule 5-0-13	The condition of an if-statement and the condition of an iteration- statement shall have type bool	MISRA 5-0-13	C++:2008	Rule
MISRA C++:2008 Rule 5-0-14	The first operand of a conditional-operator shall have type bool	MISRA 5-0-14	C++:2008	Rule
MISRA C++:2008 Rule 5-0-15	Array indexing shall be the only form of pointer arithmetic	MISRA 5-0-15	C++:2008	Rule
MISRA C++:2008 Rule 5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	MISRA 5-0-16	C++:2008	Rule
MISRA C++:2008 Rule 5-0-17	Subtraction between pointers shall only be applied to pointers that address elements of the same array	MISRA 5-0-17	C++:2008	Rule
MISRA C++:2008 Rule 5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array	MISRA 5-0-18	C++:2008	Rule
MISRA C++:2008 Rule 5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection	MISRA 5-0-19	C++:2008	Rule
MISRA C++:2008 Rule 5-0-20	Non-constant operands to a binary bitwise operator shall have the same underlying type	MISRA 5-0-20	C++:2008	Rule
MISRA C++:2008 Rule 5-0-21	Bitwise operators shall only be applied to operands of unsigned underlying type	MISRA 5-0-21	C++:2008	Rule
MISRA C++:2008 Rule 5-0-3	A cvalue expression shall not be implicitly converted to a different underlying type	MISRA 5-0-3	C++:2008	Rule
MISRA C++:2008 Rule 5-0-4	An implicit integral conversion shall not change the signedness of the underlying type	MISRA 5-0-4	C++:2008	Rule
MISRA C++:2008 Rule 5-0-5	There shall be no implicit floating-integral conversions	MISRA 5-0-5	C++:2008	Rule
MISRA C++:2008 Rule 5-0-6	An implicit integral or floating-point conversion shall not reduce the size of the underlying type	MISRA 5-0-6	C++:2008	Rule

MISRA C++:2008 Rule	Description	Polyspace Checker		
MISRA C++:2008 Rule 5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression	MISRA 5-0-7	C++:2008	Rule
MISRA C++:2008 Rule 5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression	MISRA 5-0-8	C++:2008	Rule
MISRA C++:2008 Rule 5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression	MISRA 5-0-9	C++:2008	Rule
MISRA C++:2008 Rule 5-14-1	The right hand operand of a logical && or operator shall not contain side effects	MISRA 5-14-1	C++:2008	Rule
MISRA C++:2008 Rule 5-18-1	The comma operator shall not be used	MISRA 5-18-1	C++:2008	Rule
MISRA C++:2008 Rule 5-2-1	Each operand of a logical && or shall be a postfix-expression	MISRA 5-2-1	C++:2008	Rule
MISRA C++:2008 Rule 5-2-11	The comma operator, && operator and the operator shall not be overloaded	MISRA 5-2-11	C++:2008	Rule
MISRA C++:2008 Rule 5-2-12	An identifier with array type passed as a function argument shall not decay to a pointer	MISRA 5-2-12	C++:2008	Rule
MISRA C++:2008 Rule 5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of <code>dynamic_cast</code>	MISRA 5-2-2	C++:2008	Rule
MISRA C++:2008 Rule 5-2-4	C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used	MISRA 5-2-4	C++:2008	Rule
MISRA C++:2008 Rule 5-2-5	A cast shall not remove any const or volatile qualification from the type of a pointer or reference	MISRA 5-2-5	C++:2008	Rule
MISRA C++:2008 Rule 5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type	MISRA 5-2-6	C++:2008	Rule

MISRA C++:2008 Rule	Description	Polyspace Checker		
MISRA C++:2008 Rule 5-2-7	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly	MISRA 5-2-7	C++:2008	Rule
MISRA C++:2008 Rule 5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type	MISRA 5-2-8	C++:2008	Rule
MISRA C++:2008 Rule 5-3-1	Each operand of the ! operator, the logical && or the logical operators shall have type bool	MISRA 5-3-1	C++:2008	Rule
MISRA C++:2008 Rule 5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned	MISRA 5-3-2	C++:2008	Rule
MISRA C++:2008 Rule 5-3-3	The unary & operator shall not be overloaded	MISRA 5-3-3	C++:2008	Rule
MISRA C++:2008 Rule 5-3-4	Evaluation of the operand to the sizeof operator shall not contain side effects	MISRA 5-3-4	C++:2008	Rule
MISRA C++:2008 Rule 5-8-1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand	MISRA 5-8-1	C++:2008	Rule
MISRA C++:2008 Rule 6-2-1	Assignment operators shall not be used in sub-expressions	MISRA 6-2-1	C++:2008	Rule
MISRA C++:2008 Rule 6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality	MISRA 6-2-2	C++:2008	Rule
MISRA C++:2008 Rule 6-2-3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character	MISRA 6-2-3	C++:2008	Rule
MISRA C++:2008 Rule 6-3-1	The statement forming the body of a switch, while, do while or for statement shall be a compound statement	MISRA 6-3-1	C++:2008	Rule

MISRA C++:2008 Rule	Description	Polyspace Checker		
MISRA C++:2008 Rule 6-4-1	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement	MISRA 6-4-1	C++:2008	Rule
MISRA C++:2008 Rule 6-4-2	All if Æ; else if constructs shall be terminated with an else clause	MISRA 6-4-2	C++:2008	Rule
MISRA C++:2008 Rule 6-4-3	A switch statement shall be a well-formed switch statement	MISRA 6-4-3	C++:2008	Rule
MISRA C++:2008 Rule 6-4-4	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	MISRA 6-4-4	C++:2008	Rule
MISRA C++:2008 Rule 6-4-5	An unconditional throw or break statement shall terminate every non - empty switch-clause	MISRA 6-4-5	C++:2008	Rule
MISRA C++:2008 Rule 6-4-6	The final clause of a switch statement shall be the default-clause	MISRA 6-4-6	C++:2008	Rule
MISRA C++:2008 Rule 6-4-7	The condition of a switch statement shall not have bool type	MISRA 6-4-7	C++:2008	Rule
MISRA C++:2008 Rule 6-4-8	Every switch statement shall have at least one case-clause	MISRA 6-4-8	C++:2008	Rule
MISRA C++:2008 Rule 6-5-1	A for loop shall contain a single loop-counter which shall not have floating type	MISRA 6-5-1	C++:2008	Rule
MISRA C++:2008 Rule 6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=	MISRA 6-5-2	C++:2008	Rule
MISRA C++:2008 Rule 6-5-3	The loop-counter shall not be modified within condition or statement	MISRA 6-5-3	C++:2008	Rule
MISRA C++:2008 Rule 6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop	MISRA 6-5-4	C++:2008	Rule

MISRA C++:2008 Rule	Description	Polyspace Checker		
MISRA C++:2008 Rule 6-5-5	A loop-control-variable other than the loop-counter shall not be modified within condition or expression	MISRA 6-5-5	C++:2008	Rule
MISRA C++:2008 Rule 6-5-6	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool	MISRA 6-5-6	C++:2008	Rule
MISRA C++:2008 Rule 6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement	MISRA 6-6-1	C++:2008	Rule
MISRA C++:2008 Rule 6-6-2	The goto statement shall jump to a label declared later in the same function body	MISRA 6-6-2	C++:2008	Rule
MISRA C++:2008 Rule 6-6-3	The continue statement shall only be used within a well-formed for loop	MISRA 6-6-3	C++:2008	Rule
MISRA C++:2008 Rule 6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination	MISRA 6-6-4	C++:2008	Rule
MISRA C++:2008 Rule 6-6-5	A function shall have a single point of exit at the end of the function	MISRA 6-6-5	C++:2008	Rule
MISRA C++:2008 Rule 7-1-1	A variable which is not modified shall be const qualified	MISRA 7-1-1	C++:2008	Rule
MISRA C++:2008 Rule 7-1-2	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified	MISRA 7-1-2	C++:2008	Rule
MISRA C++:2008 Rule 7-2-1	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	MISRA 7-2-1	C++:2008	Rule
MISRA C++:2008 Rule 7-3-1	The global namespace shall only contain main, namespace declarations and extern "C" declarations	MISRA 7-3-1	C++:2008	Rule
MISRA C++:2008 Rule 7-3-2	The identifier main shall not be used for a function other than the global function main	MISRA 7-3-2	C++:2008	Rule

MISRA C++:2008 Rule	Description	Polyspace Checker		
MISRA C++:2008 Rule 7-3-3	There shall be no unnamed namespaces in header files	MISRA 7-3-3	C++:2008	Rule
MISRA C++:2008 Rule 7-3-4	using-directives shall not be used	MISRA 7-3-4	C++:2008	Rule
MISRA C++:2008 Rule 7-3-5	Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier	MISRA 7-3-5	C++:2008	Rule
MISRA C++:2008 Rule 7-3-6	using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files	MISRA 7-3-6	C++:2008	Rule
MISRA C++:2008 Rule 7-4-2	Assembler instructions shall only be introduced using the asm declaration	MISRA 7-4-2	C++:2008	Rule
MISRA C++:2008 Rule 7-4-3	Assembly language shall be encapsulated and isolated	MISRA 7-4-3	C++:2008	Rule
MISRA C++:2008 Rule 7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function	MISRA 7-5-1	C++:2008	Rule
MISRA C++:2008 Rule 7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist	MISRA 7-5-2	C++:2008	Rule
MISRA C++:2008 Rule 7-5-3	A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference	MISRA 7-5-3	C++:2008	Rule
MISRA C++:2008 Rule 8-0-1	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively	MISRA 8-0-1	C++:2008	Rule
MISRA C++:2008 Rule 8-3-1	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments	MISRA 8-3-1	C++:2008	Rule
MISRA C++:2008 Rule 8-4-1	Functions shall not be defined using the ellipsis notation	MISRA 8-4-1	C++:2008	Rule

MISRA C++:2008 Rule	Description	Polyspace Checker		
MISRA C++:2008 Rule 8-4-2	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration	MISRA 8-4-2	C++:2008	Rule
MISRA C++:2008 Rule 8-4-3	All exit paths from a function with non-void return type shall have an explicit return statement with an expression	MISRA 8-4-3	C++:2008	Rule
MISRA C++:2008 Rule 8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &	MISRA 8-4-4	C++:2008	Rule
MISRA C++:2008 Rule 8-5-1	All variables shall have a defined value before they are used	MISRA 8-5-1	C++:2008	Rule
MISRA C++:2008 Rule 8-5-2	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures	MISRA 8-5-2	C++:2008	Rule
MISRA C++:2008 Rule 8-5-3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized	MISRA 8-5-3	C++:2008	Rule
MISRA C++:2008 Rule 9-3-1	const member functions shall not return non-const pointers or references to class-data	MISRA 9-3-1	C++:2008	Rule
MISRA C++:2008 Rule 9-3-2	Member functions shall not return non-const handles to class-data	MISRA 9-3-2	C++:2008	Rule
MISRA C++:2008 Rule 9-3-3	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const	MISRA 9-3-3	C++:2008	Rule
MISRA C++:2008 Rule 9-5-1	Unions shall not be used	MISRA 9-5-1	C++:2008	Rule
MISRA C++:2008 Rule 9-6-2	Bit-fields shall be either bool type or an explicitly unsigned or signed integral type	MISRA 9-6-2	C++:2008	Rule
MISRA C++:2008 Rule 9-6-3	Bit-fields shall not have enum type	MISRA 9-6-3	C++:2008	Rule
MISRA C++:2008 Rule 9-6-4	Named bit-fields with signed integer type shall have a length of more than one bit	MISRA 9-6-4	C++:2008	Rule

Unsupported Rules

Polyspace does not supports these **Required** rules:

Rule	Description
5-17-1	The semantic equivalence between a binary operator and its assignment operator form shall be preserved.
14-7-1	All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.
14-7-2	For any given template specialization, an explicit instantiation of the template with the template-arguments used in the specialization shall not render the program ill-formed.

See Also

Check MISRA C++:2008 (-misra-cpp)

More About

- “Check for and Review Coding Standard Violations” on page 16-2
- “Coding Standards”
- “Required AUTOSAR C++14 Coding Rules Supported by Polyspace Bug Finder” on page 17-125
- “Statically Enforceable AUTOSAR C++14 Rules Supported by Polyspace Bug Finder” on page 17-156
- “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78

JSF AV C++ Coding Rules

Supported JSF C++ Coding Rules

Code Size and Complexity

N.	JSF++ Definition	Polyspace Implementation
1	Any one function (or method) will contain no more than 200 logical source lines of code (L-SLOCs).	Message in report file: <function name> has <num> logical source lines of code.
3	All functions shall have a cyclomatic complexity number of 20 or less.	Message in report file: <function name> has cyclomatic complexity number equal to <num>.

Environment

N.	JSF++ Definition	Polyspace Implementation
8	All code shall conform to ISO/IEC 14882:2002(E) standard C++.	Reports the compilation error message
9	Only those characters specified in the C++ basic source character set will be used.	
11	Trigraphs will not be used.	
12	The following digraphs will not be used: <%, %>, <:, :>, %:, %:~:.	Message in report file: The following digraph will not be used: <digraph>. Reports the digraph. If the rule level is set to warning, the digraph will be allowed even if it is not supported in -compiler iso.
13	Multi-byte characters and wide string literals will not be used.	Report L'c', L"string", and use of wchar_t.
14	Literal suffixes shall use uppercase rather than lowercase letters.	
15	Provision shall be made for run-time checking (defensive programming).	Done with checks in the software.

Libraries

N.	JSF++ Definition	Polyspace Implementation
17	The error indicator errno shall not be used.	errno should not be used as a macro or a global with external "C" linkage.
18	The macro offsetof, in library <stddef.h>, shall not be used.	offsetof should not be used as a macro or a global with external "C" linkage.

N.	JSF++ Definition	Polyspace Implementation
19	<locale.h> and the <code>setlocale</code> function shall not be used.	<code>setlocale</code> and <code>localeconv</code> should not be used as a macro or a global with external "C" linkage.
20	The <code>setjmp</code> macro and the <code>longjmp</code> function shall not be used.	<code>setjmp</code> and <code>longjmp</code> should not be used as a macro or a global with external "C" linkage.
21	The signal handling facilities of <signal.h> shall not be used.	<code>signal</code> and <code>raise</code> should not be used as a macro or a global with external "C" linkage.
22	The input/output library <stdio.h> shall not be used.	all standard functions of <stdio.h> should not be used as a macro or a global with external "C" linkage.
23	The library functions <code>atof</code> , <code>atoi</code> and <code>atol</code> from library <stdlib.h> shall not be used.	<code>atof</code> , <code>atoi</code> and <code>atol</code> should not be used as a macro or a global with external "C" linkage.
24	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> from library <stdlib.h> shall not be used.	<code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> should not be used as a macro or a global with external "C" linkage.
25	The time handling functions of library <time.h> shall not be used.	<code>clock</code> , <code>difftime</code> , <code>mktime</code> , <code>asctime</code> , <code>ctime</code> , <code>gmtime</code> , <code>localtime</code> and <code>strftime</code> should not be used as a macro or a global with external "C" linkage.

Pre-Processing Directives

N.	JSF++ Definition	Polyspace Implementation
26	Only the following preprocessor directives shall be used: <code>#ifndef</code> , <code>#define</code> , <code>#endif</code> , <code>#include</code> .	
27	<code>#ifndef</code> , <code>#define</code> and <code>#endif</code> will be used to prevent multiple inclusions of the same header file. Other techniques to prevent the multiple inclusions of header files will not be used.	Detects the patterns <code>#if !defined</code> , <code>#pragma once</code> , <code>#ifdef</code> , and missing <code>#define</code> .
28	The <code>#ifndef</code> and <code>#endif</code> preprocessor directives will only be used as defined in AV Rule 27 to prevent multiple inclusions of the same header file.	Detects any use that does not comply with AV Rule 27. Assuming 35/27 is not violated, reports only <code>#ifndef</code> .
29	The <code>#define</code> preprocessor directive shall not be used to create inline macros. Inline functions shall be used instead.	Rule is split into two parts: the definition of a macro function (29.def) and the call of a macrofunction (29.use). Messages in report file: <ul style="list-style-type: none"> • 29.1 : The <code>#define</code> preprocessor directive shall not be used to create inline macros. • 29.2 : Inline functions shall be used instead of inline macros.

N.	JSF++ Definition	Polyspace Implementation
30	The <code>#define</code> preprocessor directive shall not be used to define constant values. Instead, the <code>const</code> qualifier shall be applied to variable declarations to specify constant values.	Reports <code>#define</code> of simple constants.
31	The <code>#define</code> preprocessor directive will only be used as part of the technique to prevent multiple inclusions of the same header file.	Detects use of <code>#define</code> that are not used to guard for multiple inclusion, assuming that rules 35 and 27 are not violated.
32	The <code>#include</code> preprocessor directive will only be used to include header (*.h) files.	

Header Files

N.	JSF++ Definition	Polyspace Implementation
33	The <code>#include</code> directive shall use the <code><filename.h></code> notation to include header files.	
35	A header file will contain a mechanism that prevents multiple inclusions of itself.	
39	Header files (*.h) will not contain non-const variable definitions or function definitions.	Reports definitions of global variables / function in header.

Style

N.	JSF++ Definition	Polyspace Implementation
40	Every implementation file shall include the header files that uniquely define the inline functions, types, and templates used.	Reports when type, template, or inline function is defined in source file.
41	Source lines will be kept to a length of 120 characters or less.	Polyspace ignores the newline character (<code>\n</code>) when counting the line length.
42	Each expression-statement will be on a separate line.	Reports when two consecutive expression statements are on the same line (unless the statements are part of a macro definition).
43	Tabs should be avoided.	
44	All indentations will be at least two spaces and be consistent within the same source file.	Reports when a statement indentation is not at least two spaces more than the statement containing it. Does not report bad indentation between opening braces following <code>if/else</code> , <code>do/while</code> , <code>for</code> , and <code>while</code> statements. NB: in final release it will accept any indentation
46	User-specified identifiers (internal and external) will not rely on significance of more than 64 characters.	<i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Analysis" on page 11-78.</i>
47	Identifiers will not begin with the underscore character <code>'_'</code> .	

N.	JSF++ Definition	Polyspace Implementation
48	Identifiers will not differ by: <ul style="list-style-type: none"> • Only a mixture of case • The presence/absence of the underscore character • The interchange of the letter 'O'; with the number '0' or the letter 'D' • The interchange of the letter 'I'; with the number '1' or the letter 'l' • The interchange of the letter 'S' with the number '5' • The interchange of the letter 'Z' with the number 2 • The interchange of the letter 'n' with the letter 'h' 	Checked regardless of scope. Not checked between macros and other identifiers. Messages in report file: <ul style="list-style-type: none"> • Identifier <i>Idf1 (file1.cpp line l1 column c1)</i> and <i>Idf2 (file2.cpp line l2 column c2)</i> only differ by the presence/absence of the underscore character. • Identifier <i>Idf1 (file1.cpp line l1 column c1)</i> and <i>Idf2 (file2.cpp line l2 column c2)</i> only differ by a mixture of case. • Identifier <i>Idf1 (file1.cpp line l1 column c1)</i> and <i>Idf2 (file2.cpp line l2 column c2)</i> only differ by letter 0, with the number 0.
50	The first word of the name of a class, structure, namespace, enumeration, or type created with typedef will begin with an uppercase letter. All others letters will be lowercase.	Messages in report file: <ul style="list-style-type: none"> • The first word of the name of a class will begin with an uppercase letter. • The first word of the namespace of a class will begin with an uppercase letter.
51	All letters contained in function and variables names will be composed entirely of lowercase letters.	Messages in report file: <ul style="list-style-type: none"> • All letters contained in variable names will be composed entirely of lowercase letters. • All letters contained in function names will be composed entirely of lowercase letters.
52	Identifiers for constant and enumerator values shall be lowercase.	Messages in report file: <ul style="list-style-type: none"> • Identifier for enumerator value shall be lowercase. • Identifier for template constant parameter shall be lowercase.
53	Header files will always have file name extension of ".h".	.H is allowed if you set the option -dos.
53.1	The following character sequences shall not appear in header file names: ', \, /*, //, or ".	
54	Implementation files will always have a file name extension of ".cpp".	Not case sensitive if you set the option -dos.
57	The public, protected, and private sections of a class will be declared in that order.	

N.	JSF++ Definition	Polyspace Implementation
58	When declaring and defining functions with more than two parameters, the leading parenthesis and the first argument will be written on the same line as the function name. Each additional argument will be written on a separate line (with the closing parenthesis directly after the last argument).	Detects that two parameters are not on the same line, The first parameter should be on the same line as function name. Does not check for the closing parenthesis.
59	The statements forming the body of an if, else if, else, while, do ... while or for statement shall always be enclosed in braces, even if the braces form an empty block.	<p>Messages in report file:</p> <ul style="list-style-type: none"> • The statements forming the body of an if statement shall always be enclosed in braces. • The statements forming the body of an else statement shall always be enclosed in braces. • The statements forming the body of a while statement shall always be enclosed in braces. • The statements forming the body of a do ... while statement shall always be enclosed in braces. • The statements forming the body of a for statement shall always be enclosed in braces.
60	Braces ("{}") which enclose a block will be placed in the same column, on separate lines directly before and after the block.	Detects that statement-block braces should be in the same columns.
61	Braces ("{}") which enclose a block will have nothing else on the line except comments.	
62	The dereference operator '*' and the address-of operator '&' will be directly connected with the type-specifier.	Reports when there is a space between type and "*" "&" for variables, parameters and fields declaration.
63	Spaces will not be used around '.', or '->', nor between unary operators and operands.	<p>Reports when the following characters are not directly connected to a white space:</p> <ul style="list-style-type: none"> • . • -> • ! • ~ • - • ++ • — <p>Note that a violation will be reported for "." used in float/double definition.</p>

Classes

N.	JSF++ Definition	Polyspace Implementation
67	Public and protected data should only be used in structs - not classes.	
68	Unneeded implicitly generated member functions shall be explicitly disallowed.	Reports when default constructor, assignment operator, copy constructor or destructor is not declared.
71.1	A class's virtual functions shall not be invoked from its destructor or any of its constructors.	Reports when a constructor or destructor directly calls a virtual function.
74	Initialization of nonstatic class members will be performed through the member initialization list rather than through assignment in the body of a constructor.	All data should be initialized in the initialization list except for array. Does not report that an assignment exists in ctor body. Message in report file: Initialization of nonstatic class members "<field>" will be performed through the member initialization list.
75	Members of the initialization list shall be listed in the order in which they are declared in the class.	
76	A copy constructor and an assignment operator shall be declared for classes that contain pointers to data items or nontrivial destructors.	Messages in report file: <ul style="list-style-type: none"> • no copy constructor and no copy assign • no copy constructor • no copy assign
77.1	The definition of a member function shall not contain default arguments that produce a signature identical to that of the implicitly-declared copy constructor for the corresponding class/structure.	Does not report when an explicit copy constructor exists.
78	All base classes with a virtual function shall define a virtual destructor.	
79	All resources acquired by a class shall be released by the class's destructor.	Reports when the number of "new" called in a constructor is greater than the number of "delete" called in its destructor. Note A violation is raised even if "new" is done in a "if/else".

N.	JSF++ Definition	Polyspace Implementation
81	The assignment operator shall handle self-assignment correctly	<p>Reports when copy assignment body does not begin with "if (this != arg)"</p> <p>A violation is not raised if an empty else statement follows the if, or the body contains only a return statement.</p> <p>A violation is raised when the if statement is followed by a statement other than the return statement.</p>
82	An assignment operator shall return a reference to <code>*this</code> .	<p>The following operators should return <code>*this</code> on method, and <code>*first_arg</code> on plain function:</p> <ul style="list-style-type: none"> • operator= • operator+= • operator-= • operator*= • operator >>= • operator <<= • operator /= • operator %= • operator = • operator &= • operator ^= • Prefix operator++ • Prefix operator-- <p>Does not report when no return exists.</p> <p>No special message if type does not match.</p> <p>Messages in report file:</p> <ul style="list-style-type: none"> • An assignment operator shall return a reference to <code>*this</code>. • An assignment operator shall return a reference to its first arg.
83	An assignment operator shall assign all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).	Reports when a copy assignment does not assign all data members. In a derived class, it also reports when a copy assignment does not call inherited copy assignments.

N.	JSF++ Definition	Polyspace Implementation
88	Multiple inheritance shall only be allowed in the following restricted form: n interfaces plus m private implementations, plus at most one protected implementation.	Messages in report file: <ul style="list-style-type: none"> • Multiple inheritance on public implementation shall not be allowed: <code><public_base_class></code> is not an interface. • Multiple inheritance on protected implementation shall not be allowed : <code><protected_base_class_1></code>. • <code><protected_base_class_2></code> are not interfaces.
88.1	A stateful virtual base shall be explicitly declared in each derived class that accesses it.	
89	A base class shall not be both virtual and nonvirtual in the same hierarchy.	
94	An inherited nonvirtual function shall not be redefined in a derived class.	Does not report for destructor. Message in report file: Inherited nonvirtual function %s shall not be redefined in a derived class.
95	An inherited default parameter shall never be redefined.	
96	Arrays shall not be treated polymorphically.	Reports pointer arithmetic and array like access on expressions whose pointed type is used as a base class.
97	Arrays shall not be used in interface.	Only to prevent array-to-pointer-decay. Not checked on private methods
97.1	Neither operand of an equality operator (== or !=) shall be a pointer to a virtual member function.	Reports == and != on pointer to member function of polymorphic classes (cannot determine statically if it is virtual or not), except when one argument is the null constant.

Namespaces

N.	JSF++ Definition	Polyspace Implementation
98	Every nonlocal name, except <code>main()</code> , should be placed in some namespace.	
99	Namespaces will not be nested more than two levels deep.	

Templates

N.	JSF++ Definition	Polyspace Implementation
104	A template specialization shall be declared before its use.	Reports the actual compilation error message.

Functions

N.	JSF++ Definition	Polyspace Implementation
107	Functions shall always be declared at file scope.	
108	Functions with variable numbers of arguments shall not be used.	
109	A function definition should not be placed in a class specification unless the function is intended to be inlined.	Reports when "inline" is not in the definition of a member function inside the class definition.
110	Functions with more than 7 arguments will not be used.	
111	A function shall not return a pointer or reference to a non-static local object.	Simple cases without alias effect detected.
113	Functions will have a single exit point.	Reports first return, or once per function.
114	All exit points of value-returning functions shall be through return statements.	
116	Small, concrete-type arguments (two or three words in size) should be passed by value if changes made to formal parameters should not be reflected in the calling function.	Report constant parameters references with <code>sizeof <= 2 * sizeof(int)</code> . Does not report for copy-constructor.
117	<p>Arguments should be passed by reference if NULL values are not possible:</p> <ul style="list-style-type: none"> • 117.1: An object should be passed as <code>const T&</code> if the function should not change the value of the object. • 117.2: An object should be passed as <code>T&</code> if the function may change the value of the object. 	<p>The checker flags a parameter passed by pointer if the parameter is not compared against <code>NULL</code> or <code>nullptr</code> in the function body. The absence of a check for null indicates that the parameter cannot be null and therefore can be passed by reference.</p> <p>The checker does not raise a violation:</p> <ul style="list-style-type: none"> • If a parameter is passed using a smart pointer. Only raw pointers are considered. • If the pointer parameter is not dereferenced within the function.
119	Functions shall not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).	<p>The checker reports each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported.</p> <p>You can calculate the total number of recursion cycles using the code complexity metric <code>Number of Recursions</code>. Note that unlike the checker, the metric also considers implicit calls, for instance, to compiler-generated constructors during object creation.</p>
121	Only functions with 1 or 2 statements should be considered candidates for inline functions.	Reports inline functions with more than 2 statements.

N.	JSF++ Definition	Polyspace Implementation
122	Trivial accessor and mutator functions should be inlined.	<p>The checker uses the following criteria to determine if a method is trivial:</p> <ul style="list-style-type: none"> An accessor method is trivial if it has no parameters and contains one <code>return</code> statement that returns a non-static data member or a reference to a non-static data member. <p>The return type of the method must exactly match or be a reference to the type of the data member.</p> <ul style="list-style-type: none"> A mutator method is trivial if it has a <code>void</code> return type, one parameter and contains one assignment statement that assigns the parameter to a non-static data member. <p>The parameter type must exactly match or be a reference to the type of the data member.</p> <p>The checker reports trivial accessor and mutator methods defined outside their classes without the <code>inline</code> keyword.</p> <p>The checker does not flag template methods or virtual methods.</p>

Comments

N.	JSF++ Definition	Polyspace Implementation
126	Only valid C++ style comments (<code>//</code>) shall be used.	

N.	JSF++ Definition	Polyspace Implementation
127	Code that is not used (commented out) shall be deleted.	<p>The checker uses internal heuristics to detect commented out code. For instance, characters such as #, ;, { or } indicate comments that might potentially contain code. These comments are then evaluated against other metrics to determine the likelihood of code masquerading as comment. For instance, several successive words without a symbol in between reduces this likelihood.</p> <p>The checker does not flag the following comments even if they contain code:</p> <ul style="list-style-type: none"> • Doxygen comments beginning with <code>/**, /*!, /// or //!</code>. • Comments that repeat the same symbol several times, for instance, the symbol = here: <pre data-bbox="906 829 1442 913"> // ===== // A comment // =====*/ </pre> • Comments on the first line of a file. • Comments that mix the C style <code>/* */</code> and C++ style <code>//</code>. <p>The checker considers that these comments are meant for documentation purposes or entered deliberately with some forethought.</p>
133	Every source file will be documented with an introductory comment that provides information on the file name, its contents, and any program-required information (e.g. legal statements, copyright information, etc).	<p>Reports when a file does not begin with two comment lines.</p> <p>Note: This rule cannot be annotated in the source code.</p>

Declarations and Definitions

N.	JSF++ Definition	Polyspace Implementation
135	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	

N.	JSF++ Definition	Polyspace Implementation
136	Declarations should be at the smallest feasible scope.	<p>Reports when:</p> <ul style="list-style-type: none"> • A global variable is used in only one function. • A local variable is not used in a statement (expr, return, init ...) of the same level of its declaration (in the same block) or is not used in two sub-statements of its declaration. <hr/> <p>Note</p> <ul style="list-style-type: none"> • Non-used variables are reported. • Initializations at definition are ignored (not considered an access)
137	All declarations at file scope should be static where possible.	<p>Starting in R2021a, this checker is raised on declarations of nonstatic objects that you use in only one file. The checker is raised even if you analyze a single file. The checker is not raised on the declarations of objects that remain unused, such as:</p> <ul style="list-style-type: none"> • Noninstantiated templates • Uncalled <code>static</code> or <code>extern</code> functions • Uncalled and undefined local functions • Unused types and variables <p><i>This checker is deactivated in a default Polyspace as You Code analysis. See “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78.</i></p>
138	Identifiers shall not simultaneously have both internal and external linkage in the same translation unit.	
139	External objects will not be declared in more than one file.	Reports all duplicate declarations inside a translation unit. Reports when the declaration localization is not the same in all translation units.
140	The register storage class specifier shall not be used.	
141	A class, structure, or enumeration will not be declared in the definition of its type.	

Initialization

N.	JSF++ Definition	Polyspace Implementation
142	All variables shall be initialized before use.	Polyspace reports a violation of this rule if your code contains these issues: <ul style="list-style-type: none"> • Non-initialized variable • Member not initialized in constructor • Non-initialized pointer
144	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	This covers partial initialization.
145	In an enumerator list, the '=' construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	Generates one report for an enumerator list.

Types

N.	JSF++ Definition	Polyspace Implementation
147	The underlying bit representations of floating point numbers shall not be used in any way by the programmer.	Reports on casts with float pointers (except with void*).
148	Enumeration types shall be used instead of integer types (and constants) to select from a limited series of choices.	Reports when non enumeration types are used in switches.

Constants

N.	JSF++ Definition	Polyspace Implementation
149	Octal constants (other than zero) shall not be used.	
150	Hexadecimal constants will be represented using all uppercase letters.	
151	Numeric values in code will not be used; symbolic values will be used instead.	Reports direct numeric constants (except integer/float value 1, 0) in expressions, non - const initializations. and switch cases. char constants are allowed. Does not report on templates non-type parameter.
151.1	A string literal shall not be modified.	The rule checker flags assignment of string literals to: <ul style="list-style-type: none"> • Pointers other than pointers to const objects. • Arrays that are not const-qualified.

Variables

N.	JSF++ Definition	Polyspace Implementation
152	Multiple variable declarations shall not be allowed on the same line.	Reports when two consecutive declaration statements are on the same line (unless the statements are part of a macro definition).

Unions and Bit Fields

N.	JSF++ Definition	Polyspace Implementation
153	Unions shall not be used.	
154	Bit-fields shall have explicitly unsigned integral or enumeration types only.	
156	All the members of a structure (or class) shall be named and shall only be accessed via their names.	Reports unnamed bit-fields (unnamed fields are not allowed).

Operators

N.	JSF++ Definition	Polyspace Implementation
157	The right hand operand of a && or operator shall not contain side effects.	Assumes rule 159 is not violated. Messages in report file: <ul style="list-style-type: none"> The right hand operand of a && operator shall not contain side effects. The right hand operand of a operator shall not contain side effects.
158	The operands of a logical && or shall be parenthesized if the operands contain binary operators.	Messages in report file: <ul style="list-style-type: none"> The operands of a logical && shall be parenthesized if the operands contain binary operators. The operands of a logical shall be parenthesized if the operands contain binary operators. Exception for: X Y Z , Z&&Y &&Z
159	Operators , &&, and unary & shall not be overloaded.	Messages in report file: <ul style="list-style-type: none"> Unary operator & shall not be overloaded. Operator shall not be overloaded. Operator && shall not be overloaded.
160	An assignment expression shall be used only as the expression in an expression statement.	Only simple assignment, not +=, ++, etc.
162	Signed and unsigned values shall not be mixed in arithmetic or comparison operations.	
163	Unsigned arithmetic shall not be used.	

N.	JSF++ Definition	Polyspace Implementation
164	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the left-hand operand (inclusive).	
164.1	The left-hand operand of a right-shift operator shall not have a negative value.	Detects constant case +. Found by the software for dynamic cases.
165	The unary minus operator shall not be applied to an unsigned expression.	
166	The sizeof operator will not be used on expressions that contain side effects.	
168	The comma operator shall not be used.	

Pointers and References

N.	JSF++ Definition	Polyspace Implementation
169	Pointers to pointers should be avoided when possible.	Reports second-level pointers, except for arguments of main.
170	More than 2 levels of pointer indirection shall not be used.	Only reports on variables/parameters.
171	Relational operators shall not be applied to pointer types except where both operands are of the same type and point to: <ul style="list-style-type: none"> • the same object, • the same function, • members of the same object, or • elements of the same array (including one past the end of the same array). 	Reports when relational operator are used on pointer types (casts ignored).
173	The address of an object with automatic storage shall not be assigned to an object which persists after the object has ceased to exist.	
174	The null pointer shall not be de-referenced.	Done with checks in software.
175	A pointer shall not be compared to NULL or be assigned NULL; use plain 0 instead.	Reports usage of NULL macro in pointer contexts.
176	A typedef will be used to simplify program syntax when declaring function pointers.	Reports non-typedef function pointers, or pointers to member functions for types of variables, fields, parameters. Returns type of function, cast, and exception specification.

Type Conversions

N.	JSF++ Definition	Polyspace Implementation
177	User-defined conversion functions should be avoided.	Reports user defined conversion function, non-explicit constructor with one parameter or default value for others (even undefined ones). Does not report copy-constructor. Additional message for constructor case: This constructor should be flagged as "explicit".
178	Down casting (casting from base to derived class) shall only be allowed through one of the following mechanism: <ul style="list-style-type: none"> Virtual functions that act like dynamic casts (most likely useful in relatively simple cases). Use of the visitor (or similar) pattern (most likely useful in complicated cases). 	Reports explicit down casting, <code>dynamic_cast</code> included. (Visitor patter does not have a special case.)
179	A pointer to a virtual base class shall not be converted to a pointer to a derived class.	Reports this specific down cast. Allows <code>dynamic_cast</code> .
180	Implicit conversions that may result in a loss of information shall not be used.	Reports the following implicit casts : <code>integer => smaller integer unsigned => smaller or eq signed signed => smaller or eq un-signed integer => float float => integer</code> Does not report for cast to <code>bool</code> reports for implicit cast on constant done with the option <code>-scalar-overflows-checks signed-and-unsigned</code>
181	Redundant explicit casts will not be used.	Reports useless cast: <code>cast T to T</code> . Casts to equivalent typedefs are also reported.
182	Type casting from any type to or from pointers shall not be used.	Does not report when Rule 181 applies.
184	Floating point numbers shall not be converted to integers unless such a conversion is a specified algorithmic requirement or is necessary for a hardware interface.	Reports <code>float->int</code> conversions. Does not report implicit ones.
185	C++ style casts (<code>const_cast</code> , <code>reinterpret_cast</code> , and <code>static_cast</code>) shall be used instead of the traditional C-style casts.	

Flow Control Standards

N.	JSF++ Definition	Polyspace Implementation
186	There shall be no unreachable code.	Done with gray checks in the software.

N.	JSF++ Definition	Polyspace Implementation
187	All non-null statements shall potentially have a side-effect.	
188	Labels will not be used, except in switch statements.	
189	The <code>goto</code> statement shall not be used.	
190	The <code>continue</code> statement shall not be used.	
191	The <code>break</code> statement shall not be used (except to terminate the cases of a switch statement).	
192	All <code>if, else if</code> constructs will contain either a final <code>else</code> clause or a comment indicating why a final <code>else</code> clause is not necessary.	else if should contain an else clause.
193	Every non-empty <code>case</code> clause in a switch statement shall be terminated with a <code>break</code> statement.	
194	All switch statements that do not intend to test for every enumeration value shall contain a final <code>default</code> clause.	Reports only for missing default.
195	A switch expression will not represent a Boolean value.	
196	Every switch statement will have at least two cases and a potential <code>default</code> .	
197	Floating point variables shall not be used as loop counters.	Assumes 1 loop parameter.
198	The initialization expression in a <code>for</code> loop will perform no actions other than to initialize the value of a single <code>for</code> loop parameter.	Reports if loop parameter cannot be determined. Assumes Rule 200 is not violated. The loop variable parameter is assumed to be a variable.
199	The increment expression in a <code>for</code> loop will perform no action other than to change a single loop parameter to the next value for the loop.	Assumes 1 loop parameter (Rule 198), with non class type. Rule 200 must not be violated for this rule to be reported.
200	Null initialize or increment expressions in <code>for</code> loops will not be used; a <code>while</code> loop will be used instead.	
201	Numeric variables being used within a <code>for</code> loop for iteration counting shall not be modified in the body of the loop.	Assumes 1 loop parameter (AV rule 198), and no alias writes.

Expressions

N.	JSF++ Definition	Polyspace Implementation
202	Floating point variables shall not be tested for exact equality or inequality.	Reports only direct equality/inequality. Check done for all expressions.
203	Evaluation of expressions shall not lead to overflow/underflow.	Done with overflow checks in the software.

N.	JSF++ Definition	Polyspace Implementation
204	<p>A single operation with side-effects shall only be used in the following contexts:</p> <ul style="list-style-type: none"> • by itself • the right-hand side of an assignment • a condition • the only argument expression with a side-effect in a function call • condition of a loop • switch condition • single part of a chained operation 	<p>Reports when:</p> <ul style="list-style-type: none"> • A side effect is found in a return statement • A side effect exists on a single value, and only one operand of the function call has a side effect.
204.1	<p>The value of an expression shall be the same under any order of evaluation that the standard permits.</p>	<p>Reports when:</p> <ul style="list-style-type: none"> • A variable is written and reused within the same expression. • A volatile variable is accessed more than once.
205	<p>The volatile keyword shall not be used unless directly interfacing with hardware.</p>	<p>Reports if volatile keyword is used.</p>

Memory Allocation

N.	JSF++ Definition	Polyspace Implementation
206	<p>Allocation/deallocation from/to the free store (heap) shall not occur after initialization.</p>	<p>Reports calls to C library functions: <code>malloc / calloc / realloc / free</code> and all <code>new/delete</code> operators in functions or methods.</p>

Fault Handling

N.	JSF++ Definition	Polyspace Implementation
208	<p>C++ exceptions shall not be used.</p>	<p>Reports <code>try, catch, throw spec, and throw.</code></p>

Portable Code

N.	JSF++ Definition	Polyspace Implementation
209	<p>The basic types of <code>int, short, long, float</code> and <code>double</code> shall not be used, but specific-length equivalents should be <code>typedef</code>'d accordingly for each compiler, and these type names used in the code.</p>	<p>Only allows use of basic types through direct <code>typedefs</code>.</p>
213	<p>No dependence shall be placed on C++'s operator precedence rules, below arithmetic operators, in expressions.</p>	<p>Reports when a binary operation has one operand that is not parenthesized and is an operation with inferior precedence level.</p> <p>Reports bitwise and shifts operators that are used without parenthesis and binary operation arguments.</p>

N.	JSF++ Definition	Polyspace Implementation
215	Pointer arithmetic will not be used.	Reports: p + Ip - Ip++p--p+=p-= Allows p[i].

Unsupported JSF++ Rules

- “Code Size and Complexity” on page 17-118
- “Rules” on page 17-118
- “Environment” on page 17-119
- “Libraries” on page 17-119
- “Header Files” on page 17-119
- “Style” on page 17-119
- “Classes” on page 17-120
- “Namespaces” on page 17-121
- “Templates” on page 17-121
- “Functions” on page 17-121
- “Comments” on page 17-121
- “Initialization” on page 17-122
- “Types” on page 17-122
- “Unions and Bit Fields” on page 17-122
- “Operators” on page 17-122
- “Type Conversions” on page 17-122
- “Expressions” on page 17-123
- “Memory Allocation” on page 17-123
- “Portable Code” on page 17-123
- “Efficiency Considerations” on page 17-123
- “Miscellaneous” on page 17-123
- “Testing” on page 17-124

Code Size and Complexity

N.	JSF++ Definition
2	There shall not be any self-modifying code.

Rules

N.	JSF++ Definition
4	To break a “should” rule, the following approval must be received by the developer: <ul style="list-style-type: none"> • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool)

N.	JSF++ Definition
5	To break a “will” or a “shall” rule, the following approvals must be received by the developer: <ul style="list-style-type: none"> • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool) • approval from the software product manager (obtained by the unit approval in the developmental CM tool)
6	Each deviation from a “shall” rule shall be documented in the file that contains the deviation. Deviations from this rule shall not be allowed, AV Rule 5 notwithstanding.
7	Approval will not be required for a deviation from a “shall” or “will” rule that complies with an exception specified by that rule.

Environment

N.	JSF++ Definition
10	Values of character types will be restricted to a defined and documented subset of ISO 10646 1.

Libraries

N.	JSF++ Definition
16	Only DO-178B level A [15] certifiable or SEAL 1 C/C++ libraries shall be used with safety-critical (i.e. SEAL 1) code.

Header Files

N.	JSF++ Definition
34	Header files should contain logically related declarations only.
36	Compilation dependencies should be minimized when possible.
37	Header (include) files should include only those header files that are required for them to successfully compile. Files that are only used by the associated .cpp file should be placed in the .cpp file — not the .h file.
38	Declarations of classes that are only accessed via pointers (*) or references (&) should be supplied by forward headers that contain only forward declarations.

Style

N.	JSF++ Definition
45	All words in an identifier will be separated by the ‘_’ character.
49	All acronyms in an identifier will be composed of uppercase letters.
55	The name of a header file should reflect the logical entity for which it provides declarations.
56	The name of an implementation file should reflect the logical entity for which it provides definitions and have a “.cpp” extension (this name will normally be identical to the header file that provides the corresponding declarations.) At times, more than one .cpp file for a given logical entity will be required. In these cases, a suffix should be appended to reflect a logical differentiation.

Classes

N.	JSF++ Definition
64	A class interface should be complete and minimal.
65	A structure should be used to model an entity that does not require an invariant.
66	A class should be used to model an entity that maintains an invariant.
69	A member function that does not affect the state of an object (its instance variables) will be declared const. Member functions should be const by default. Only when there is a clear, explicit reason should the const modifier on member functions be omitted.
70	A class will have friends only when a function or object requires access to the private elements of the class, but is unable to be a member of the class for logical or efficiency reasons.
70.1	An object shall not be improperly used before its lifetime begins or after its lifetime ends.
71	Calls to an externally visible operation of an object, other than its constructors, shall not be allowed until the object has been fully initialized.
72	<p>The invariant for a class should be:</p> <ul style="list-style-type: none"> • A part of the postcondition of every class constructor, • A part of the precondition of the class destructor (if any), • A part of the precondition and postcondition of every other publicly accessible operation.
73	Unnecessary default constructors shall not be defined.
77	A copy constructor shall copy all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).
80	The default copy and assignment operators will be used for classes when those operators offer reasonable semantics.
84	Operator overloading will be used sparingly and in a conventional manner.
85	When two operators are opposites (such as == and !=), both will be defined and one will be defined in terms of the other.
86	Concrete types should be used to represent simple independent concepts.
87	Hierarchies should be based on abstract classes.
90	Heavily used interfaces should be minimal, general and abstract.
91	Public inheritance will be used to implement “is-a” relationships.
92	<p>A subtype (publicly derived classes) will conform to the following guidelines with respect to all classes involved in the polymorphic assignment of different subclass instances to the same variable or parameter during the execution of the system:</p> <ul style="list-style-type: none"> • Preconditions of derived methods must be at least as weak as the preconditions of the methods they override. • Postconditions of derived methods must be at least as strong as the postconditions of the methods they override. <p>In other words, subclass methods must expect less and deliver more than the base class methods they override. This rule implies that subtypes will conform to the Liskov Substitution Principle.</p>
93	“has-a” or “is-implemented-in-terms-of” relationships will be modeled through membership or non-public inheritance.

Namespaces

N.	JSF++ Definition
100	<p>Elements from a namespace should be selected as follows:</p> <ul style="list-style-type: none"> • using declaration or explicit qualification for few (approximately five) names, • using directive for many names.

Templates

N.	JSF++ Definition
101	<p>Templates shall be reviewed as follows:</p> <ol style="list-style-type: none"> 1 with respect to the template in isolation considering assumptions or requirements placed on its arguments. 2 with respect to all functions instantiated by actual arguments.
102	Template tests shall be created to cover all actual template instantiations.
103	Constraint checks should be applied to template arguments.
105	A template definition's dependence on its instantiation contexts should be minimized.
106	Specializations for pointer types should be made where appropriate.

Functions

N.	JSF++ Definition
112	Function return values should not obscure resource ownership.
115	If a function returns error information, then that error information will be tested.
118	<p>Arguments should be passed via pointers if NULL values are possible:</p> <ul style="list-style-type: none"> • 118.1 - An object should be passed as <code>const T*</code> if its value should not be modified. • 118.2 - An object should be passed as <code>T*</code> if its value may be modified.
120	Overloaded operations or methods should form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal parameters.
123	The number of accessor and mutator functions should be minimized.
124	Trivial forwarding functions should be inlined.
125	Unnecessary temporary objects should be avoided.

Comments

N.	JSF++ Definition
128	Comments that document actions or sources (e.g. tables, figures, paragraphs, etc.) outside of the file being documented will not be allowed.
129	Comments in header files should describe the externally visible behavior of the functions or classes being documented.
130	The purpose of every line of executable code should be explained by a comment, although one comment may describe more than one line of code.

N.	JSF++ Definition
131	One should avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).
132	Each variable declaration, typedef, enumeration value, and structure member will be commented.
134	Assumptions (limitations) made by functions should be documented in the function's preamble.

Initialization

N.	JSF++ Definition
143	Variables will not be introduced until they can be initialized with meaningful values. (See also AV Rule 136, AV Rule 142, and AV Rule 73 concerning declaration scope, initialization before use, and default constructors respectively.)

Types

N.	JSF++ Definition
146	Floating point implementations shall comply with a defined floating point standard. The standard that will be used is the ANSI/IEEE® Std 754 [1].

Unions and Bit Fields

N.	JSF++ Definition
155	Bit-fields will not be used to pack data into a word for the sole purpose of saving space.

Operators

N.	JSF++ Definition
167	The implementation of integer division in the chosen compiler shall be determined, documented and taken into account.

Type Conversions

N.	JSF++ Definition
183	Every possible measure should be taken to avoid type casting.

Expressions

N.	JSF++ Definition
204	<p>A single operation with side-effects shall only be used in the following contexts:</p> <ol style="list-style-type: none"> 1 by itself 2 the right-hand side of an assignment 3 a condition 4 the only argument expression with a side-effect in a function call 5 condition of a loop 6 switch condition 7 single part of a chained operation

Memory Allocation

N.	JSF++ Definition
207	Unencapsulated global data will be avoided.

Portable Code

N.	JSF++ Definition
210	Algorithms shall not make assumptions concerning how data is represented in memory (e.g. big endian vs. little endian, base class subobject ordering in derived classes, nonstatic data member ordering across access specifiers, etc.).
210.1	Algorithms shall not make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier.
211	Algorithms shall not assume that shorts, ints, longs, floats, doubles or long doubles begin at particular addresses.
212	Underflow or overflow functioning shall not be depended on in any special way.
214	Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.

Efficiency Considerations

N.	JSF++ Definition
216	Programmers should not attempt to prematurely optimize code.

Miscellaneous

N.	JSF++ Definition
217	Compile-time and link-time errors should be preferred over run-time errors.
218	Compiler warning levels will be set in compliance with project policies.

Testing

N.	JSF++ Definition
219	All tests applied to a base class interface shall be applied to all derived class interfaces as well. If the derived class poses stronger postconditions/invariants, then the new postconditions /invariants shall be substituted in the derived class tests.
220	Structural coverage algorithms shall be applied against flattened classes.
221	Structural coverage of a class within an inheritance hierarchy containing virtual functions shall include testing every possible resolution for each set of identical polymorphic references.

Required AUTOSAR C++14 Coding Rules Supported by Polyspace Bug Finder

The AUTOSAR C++14 standard classifies the rules that compliant C++ code must follow as **Required**. Polyspace Bug Finder supports 337 out of 362 **Required** AUTOSAR C++14 coding rules.

Supported Rules

As of R2023a, Polyspace supports these **Required** rules.

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A0-1-1	A project shall not contain instances of non-volatile variables being given values that are not subsequently used	AUTOSAR A0-1-1	C++14	Rule
AUTOSAR C++14 Rule A0-1-2	The value returned by a function having a non-void return type that is not an overloaded operator shall be used	AUTOSAR A0-1-2	C++14	Rule
AUTOSAR C++14 Rule A0-1-3	Every function defined in an anonymous namespace, or static function with internal linkage, or private member function shall be used	AUTOSAR A0-1-3	C++14	Rule
AUTOSAR C++14 Rule A0-1-4	There shall be no unused named parameters in non-virtual functions	AUTOSAR A0-1-4	C++14	Rule
AUTOSAR C++14 Rule A0-1-5	There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it	AUTOSAR A0-1-5	C++14	Rule
AUTOSAR C++14 Rule A0-4-2	Type long double shall not be used	AUTOSAR A0-4-2	C++14	Rule
AUTOSAR C++14 Rule A0-4-4	Range, domain and pole errors shall be checked when using math functions	AUTOSAR A0-4-4	C++14	Rule
AUTOSAR C++14 Rule A1-1-1	All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features	AUTOSAR A1-1-1	C++14	Rule
AUTOSAR C++14 Rule A10-1-1	Class shall not be derived from more than one base class which is not an interface class	AUTOSAR A10-1-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A10-2-1	Non-virtual public or protected member functions shall not be redefined in derived classes	AUTOSAR A10-2-1	C++14	Rule
AUTOSAR C++14 Rule A10-3-1	Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final	AUTOSAR A10-3-1	C++14	Rule
AUTOSAR C++14 Rule A10-3-2	Each overriding virtual function shall be declared with the override or final specifier	AUTOSAR A10-3-2	C++14	Rule
AUTOSAR C++14 Rule A10-3-3	Virtual functions shall not be introduced in a final class	AUTOSAR A10-3-3	C++14	Rule
AUTOSAR C++14 Rule A10-3-5	A user-defined assignment operator shall not be virtual	AUTOSAR A10-3-5	C++14	Rule
AUTOSAR C++14 Rule A11-0-2	A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class	AUTOSAR A11-0-2	C++14	Rule
AUTOSAR C++14 Rule A11-3-1	Friend declarations shall not be used	AUTOSAR A11-3-1	C++14	Rule
AUTOSAR C++14 Rule A12-0-1	If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well	AUTOSAR A12-0-1	C++14	Rule
AUTOSAR C++14 Rule A12-0-2	Bitwise operations and operations that assume data representation in memory shall not be performed on objects	AUTOSAR A12-0-2	C++14	Rule
AUTOSAR C++14 Rule A12-1-1	Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members	AUTOSAR A12-1-1	C++14	Rule
AUTOSAR C++14 Rule A12-1-2	Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type	AUTOSAR A12-1-2	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A12-1-3	If all user-defined constructors of a class initialize data members with constant values that are the same across all constructors, then data members shall be initialized using NSDMI instead	AUTOSAR A12-1-3	C++14	Rule
AUTOSAR C++14 Rule A12-1-4	All constructors that are callable with a single argument of fundamental type shall be declared explicit	AUTOSAR A12-1-4	C++14	Rule
AUTOSAR C++14 Rule A12-1-5	Common class initialization for non-constant members shall be done by a delegating constructor	AUTOSAR A12-1-5	C++14	Rule
AUTOSAR C++14 Rule A12-1-6	Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors	AUTOSAR A12-1-6	C++14	Rule
AUTOSAR C++14 Rule A12-4-1	Destructor of a base class shall be public virtual, public override or protected non-virtual	AUTOSAR A12-4-1	C++14	Rule
AUTOSAR C++14 Rule A12-6-1	All class data members that are initialized by the constructor shall be initialized using member initializers	AUTOSAR A12-6-1	C++14	Rule
AUTOSAR C++14 Rule A12-7-1	If the behavior of a user-defined special member function is identical to implicitly defined special member function, then it shall be defined "=default" or be left undefined	AUTOSAR A12-7-1	C++14	Rule
AUTOSAR C++14 Rule A12-8-1	Move and copy constructors shall move and respectively copy base classes and data members of a class, without any side effects	AUTOSAR A12-8-1	C++14	Rule
AUTOSAR C++14 Rule A12-8-3	Moved-from object shall not be read-accessed	AUTOSAR A12-8-3	C++14	Rule
AUTOSAR C++14 Rule A12-8-4	Move constructor shall not initialize its class members and base classes using copy semantics	AUTOSAR A12-8-4	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A12-8-5	A copy assignment and a move assignment operators shall handle self-assignment	AUTOSAR A12-8-5	C++14	Rule
AUTOSAR C++14 Rule A12-8-6	Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class	AUTOSAR A12-8-6	C++14	Rule
AUTOSAR C++14 Rule A13-1-2	User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters	AUTOSAR A13-1-2	C++14	Rule
AUTOSAR C++14 Rule A13-1-3	User defined literals operators shall only perform conversion of passed parameters	AUTOSAR A13-1-3	C++14	Rule
AUTOSAR C++14 Rule A13-2-1	An assignment operator shall return a reference to "this"	AUTOSAR A13-2-1	C++14	Rule
AUTOSAR C++14 Rule A13-2-2	A binary arithmetic operator and a bitwise operator shall return a "prvalue"	AUTOSAR A13-2-2	C++14	Rule
AUTOSAR C++14 Rule A13-2-3	A relational operator shall return a boolean value	AUTOSAR A13-2-3	C++14	Rule
AUTOSAR C++14 Rule A13-3-1	A function that contains "forwarding reference" as its argument shall not be overloaded	AUTOSAR A13-3-1	C++14	Rule
AUTOSAR C++14 Rule A13-5-1	If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented	AUTOSAR A13-5-1	C++14	Rule
AUTOSAR C++14 Rule A13-5-2	All user-defined conversion operators shall be defined explicit	AUTOSAR A13-5-2	C++14	Rule
AUTOSAR C++14 Rule A13-5-4	If two opposite operators are defined, one shall be defined in terms of the other	AUTOSAR A13-5-4	C++14	Rule
AUTOSAR C++14 Rule A13-5-5	Comparison operators shall be non-member functions with identical parameter types and noexcept	AUTOSAR A13-5-5	C++14	Rule
AUTOSAR C++14 Rule A13-6-1	Digit sequences separators ' shall only be used as follows: (1) for decimal, every 3 digits, (2) for hexadecimal, every 2 digits, (3) for binary, every 4 digits	AUTOSAR A13-6-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A14-5-1	A template constructor shall not participate in overload resolution for a single argument of the enclosing class type	AUTOSAR A14-5-1	C++14	Rule
AUTOSAR C++14 Rule A14-7-1	A type used as a template argument shall provide all members that are used by the template	AUTOSAR A14-7-1	C++14	Rule
AUTOSAR C++14 Rule A14-7-2	Template specialization shall be declared in the same file (1) as the primary template (2) as a user-defined type, for which the specialization is declared	AUTOSAR A14-7-2	C++14	Rule
AUTOSAR C++14 Rule A14-8-2	Explicit specializations of function templates shall not be used	AUTOSAR A14-8-2	C++14	Rule
AUTOSAR C++14 Rule A15-0-2	At least the basic guarantee for exception safety shall be provided for all operations. In addition, each function may offer either the strong guarantee or the nothrow guarantee	AUTOSAR A15-0-2	C++14	Rule
AUTOSAR C++14 Rule A15-0-3	Exception safety guarantee of a called function shall be considered	AUTOSAR A15-0-3	C++14	Rule
AUTOSAR C++14 Rule A15-0-7	Exception handling mechanism shall guarantee a deterministic worst-case time execution time	AUTOSAR A15-0-7	C++14	Rule
AUTOSAR C++14 Rule A15-1-2	An exception object shall not be a pointer	AUTOSAR A15-1-2	C++14	Rule
AUTOSAR C++14 Rule A15-1-4	If a function exits with an exception, then before a throw, the function shall place all objects/resources that the function constructed in valid states or it shall delete them.	AUTOSAR A15-1-4	C++14	Rule
AUTOSAR C++14 Rule A15-1-5	Exceptions shall not be thrown across execution boundaries	AUTOSAR A15-1-5	C++14	Rule
AUTOSAR C++14 Rule A15-2-1	Constructors that are not noexcept shall not be invoked before program startup	AUTOSAR A15-2-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A15-2-2	If a constructor is not noexcept and the constructor cannot finish object initialization, then it shall deallocate the object's resources and it shall throw an exception	AUTOSAR A15-2-2	C++14	Rule
AUTOSAR C++14 Rule A15-3-3	Main function and a task main function shall catch at least: base class exceptions from all third-party libraries used, std::exception and all otherwise unhandled exceptions	AUTOSAR A15-3-3	C++14	Rule
AUTOSAR C++14 Rule A15-3-4	Catch-all (ellipsis and std::exception) handlers shall be used only in (a) main, (b) task main functions, (c) in functions that are supposed to isolate independent components and (d) when calling third-party code that uses exceptions not according to AUTOSAR C++14 guidelines	AUTOSAR A15-3-4	C++14	Rule
AUTOSAR C++14 Rule A15-3-5	A class type exception shall be caught by reference or const reference	AUTOSAR A15-3-5	C++14	Rule
AUTOSAR C++14 Rule A15-4-1	Dynamic exception-specification shall not be used	AUTOSAR A15-4-1	C++14	Rule
AUTOSAR C++14 Rule A15-4-2	If a function is declared to be noexcept, noexcept(true) or noexcept(<true condition>), then it shall not exit with an exception	AUTOSAR A15-4-2	C++14	Rule
AUTOSAR C++14 Rule A15-4-3	The noexcept specification of a function shall either be identical across all translation units, or identical or more restrictive between a virtual member function and an overrider	AUTOSAR A15-4-3	C++14	Rule
AUTOSAR C++14 Rule A15-4-4	A declaration of non-throwing function shall contain noexcept specification	AUTOSAR A15-4-4	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A15-4-5	Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overriders	AUTOSAR A15-4-5	C++14	Rule
AUTOSAR C++14 Rule A15-5-1	All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A noexcept exception specification shall be added to these functions as appropriate	AUTOSAR A15-5-1	C++14	Rule
AUTOSAR C++14 Rule A15-5-2	Program shall not be abruptly terminated. In particular, an implicit or explicit invocation of <code>std::abort()</code> , <code>std::quick_exit()</code> , <code>std::Exit()</code> , <code>std::terminate()</code> shall not be done	AUTOSAR A15-5-2	C++14	Rule
AUTOSAR C++14 Rule A15-5-3	The <code>std::terminate()</code> function shall not be called implicitly	AUTOSAR A15-5-3	C++14	Rule
AUTOSAR C++14 Rule A16-0-1	The preprocessor shall only be used for unconditional and conditional file inclusion and include guards, and using specific directives	AUTOSAR A16-0-1	C++14	Rule
AUTOSAR C++14 Rule A16-2-1	The <code>'</code> , <code>"</code> , <code>/*</code> , <code>//</code> , <code>\</code> characters shall not occur in a header file name or in <code>#include</code> directive	AUTOSAR A16-2-1	C++14	Rule
AUTOSAR C++14 Rule A16-2-2	There shall be no unused include directives	AUTOSAR A16-2-2	C++14	Rule
AUTOSAR C++14 Rule A16-2-3	An include directive shall be added explicitly for every symbol used in a file	AUTOSAR A16-2-3	C++14	Rule
AUTOSAR C++14 Rule A16-6-1	<code>#error</code> directive shall not be used	AUTOSAR A16-6-1	C++14	Rule
AUTOSAR C++14 Rule A16-7-1	The <code>#pragma</code> directive shall not be used	AUTOSAR A16-7-1	C++14	Rule
AUTOSAR C++14 Rule A17-0-1	Reserved identifiers, macros and functions in the C++ standard library shall not be defined, redefined or undefined	AUTOSAR A17-0-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A17-1-1	Use of the C Standard Library shall be encapsulated and isolated	AUTOSAR A17-1-1	C++14	Rule
AUTOSAR C++14 Rule A17-6-1	Non-standard entities shall not be added to standard namespaces	AUTOSAR A17-6-1	C++14	Rule
AUTOSAR C++14 Rule A18-0-1	The C library facilities shall only be accessed through C++ library headers	AUTOSAR A18-0-1	C++14	Rule
AUTOSAR C++14 Rule A18-0-2	The error state of a conversion from string to a numeric value shall be checked	AUTOSAR A18-0-2	C++14	Rule
AUTOSAR C++14 Rule A18-0-3	The library <locale> (locale.h) and the setlocale function shall not be used	AUTOSAR A18-0-3	C++14	Rule
AUTOSAR C++14 Rule A18-1-1	C-style arrays shall not be used	AUTOSAR A18-1-1	C++14	Rule
AUTOSAR C++14 Rule A18-1-2	The <code>std::vector<bool></code> specialization shall not be used	AUTOSAR A18-1-2	C++14	Rule
AUTOSAR C++14 Rule A18-1-3	The <code>std::auto_ptr</code> shall not be used	AUTOSAR A18-1-3	C++14	Rule
AUTOSAR C++14 Rule A18-1-4	A pointer pointing to an element of an array of objects shall not be passed to a smart pointer of single object type	AUTOSAR A18-1-4	C++14	Rule
AUTOSAR C++14 Rule A18-1-6	All <code>std::hash</code> specializations for user-defined types shall have a <code>noexcept</code> function call operator	AUTOSAR A18-1-6	C++14	Rule
AUTOSAR C++14 Rule A18-5-1	Functions <code>malloc</code> , <code>calloc</code> , <code>realloc</code> and <code>free</code> shall not be used	AUTOSAR A18-5-1	C++14	Rule
AUTOSAR C++14 Rule A18-5-10	Placement <code>new</code> shall be used only with properly aligned pointers to sufficient storage capacity	AUTOSAR A18-5-10	C++14	Rule
AUTOSAR C++14 Rule A18-5-11	"operator <code>new</code> " and "operator <code>delete</code> " shall be defined together	AUTOSAR A18-5-11	C++14	Rule
AUTOSAR C++14 Rule A18-5-2	Non-placement <code>new</code> or <code>delete</code> expressions shall not be used	AUTOSAR A18-5-2	C++14	Rule
AUTOSAR C++14 Rule A18-5-3	The form of <code>delete</code> operator shall match the form of <code>new</code> operator used to allocate the memory	AUTOSAR A18-5-3	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A18-5-4	If a project has sized or unsized version of operator 'delete' globally defined, then both sized and unsized versions shall be defined	AUTOSAR A18-5-4	C++14	Rule
AUTOSAR C++14 Rule A18-5-5	Memory management functions shall ensure the following: (a) deterministic behavior resulting with the existence of worst-case execution time, (b) avoiding memory fragmentation, (c) avoid running out of memory, (d) avoiding mismatched allocations or deallocations, (e) no dependence on non-deterministic calls to kernel	AUTOSAR A18-5-5	C++14	Rule
AUTOSAR C++14 Rule A18-5-7	If non-real-time implementation of dynamic memory management functions is used in the project, then memory shall only be allocated and deallocated during non-real-time program phases	AUTOSAR A18-5-7	C++14	Rule
AUTOSAR C++14 Rule A18-5-8	Objects that do not outlive a function shall have automatic storage duration	AUTOSAR A18-5-8	C++14	Rule
AUTOSAR C++14 Rule A18-5-9	Custom implementations of dynamic memory allocation and deallocation functions shall meet the semantic requirements specified in the corresponding "Required behaviour" clause from the C++ Standard	AUTOSAR A18-5-9	C++14	Rule
AUTOSAR C++14 Rule A18-9-1	The <code>std::bind</code> shall not be used	AUTOSAR A18-9-1	C++14	Rule
AUTOSAR C++14 Rule A18-9-2	Forwarding values to other functions shall be done via: (1) <code>std::move</code> if the value is an rvalue reference, (2) <code>std::forward</code> if the value is forwarding reference	AUTOSAR A18-9-2	C++14	Rule
AUTOSAR C++14 Rule A18-9-3	The <code>std::move</code> shall not be used on objects declared <code>const</code> or <code>const&</code>	AUTOSAR A18-9-3	C++14	Rule
AUTOSAR C++14 Rule A18-9-4	An argument to <code>std::forward</code> shall not be subsequently used	AUTOSAR A18-9-4	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A2-10-1	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope	AUTOSAR A2-10-1	C++14	Rule
AUTOSAR C++14 Rule A2-10-4	The identifier name of a non-member object with static storage duration or static function shall not be reused within a namespace	AUTOSAR A2-10-4	C++14	Rule
AUTOSAR C++14 Rule A2-10-6	A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope	AUTOSAR A2-10-6	C++14	Rule
AUTOSAR C++14 Rule A2-11-1	Volatile keyword shall not be used	AUTOSAR A2-11-1	C++14	Rule
AUTOSAR C++14 Rule A2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2014 shall be used	AUTOSAR A2-13-1	C++14	Rule
AUTOSAR C++14 Rule A2-13-2	String literals with different encoding prefixes shall not be concatenated	AUTOSAR A2-13-2	C++14	Rule
AUTOSAR C++14 Rule A2-13-3	Type wchar_t shall not be used	AUTOSAR A2-13-3	C++14	Rule
AUTOSAR C++14 Rule A2-13-4	String literals shall not be assigned to non-constant pointers	AUTOSAR A2-13-4	C++14	Rule
AUTOSAR C++14 Rule A2-13-6	Universal character names shall be used only inside character or string literals	AUTOSAR A2-13-6	C++14	Rule
AUTOSAR C++14 Rule A2-3-1	Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code	AUTOSAR A2-3-1	C++14	Rule
AUTOSAR C++14 Rule A2-5-1	Trigraphs shall not be used	AUTOSAR A2-5-1	C++14	Rule
AUTOSAR C++14 Rule A2-5-2	Digraphs shall not be used	AUTOSAR A2-5-2	C++14	Rule
AUTOSAR C++14 Rule A2-7-1	The character \ shall not occur as a last character of a C++ comment	AUTOSAR A2-7-1	C++14	Rule
AUTOSAR C++14 Rule A2-7-2	Sections of code shall not be "commented out"	AUTOSAR A2-7-2	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A2-7-3	All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation	AUTOSAR A2-7-3	C++14	Rule
AUTOSAR C++14 Rule A2-8-1	A header file name should reflect the logical entity for which it provides declarations.	AUTOSAR A2-8-1	C++14	Rule
AUTOSAR C++14 Rule A20-8-1	An already-owned pointer value shall not be stored in an unrelated smart pointer	AUTOSAR A20-8-1	C++14	Rule
AUTOSAR C++14 Rule A20-8-2	A <code>std::unique_ptr</code> shall be used to represent exclusive ownership	AUTOSAR A20-8-2	C++14	Rule
AUTOSAR C++14 Rule A20-8-3	A <code>std::shared_ptr</code> shall be used to represent shared ownership	AUTOSAR A20-8-3	C++14	Rule
AUTOSAR C++14 Rule A20-8-4	A <code>std::unique_ptr</code> shall be used over <code>std::shared_ptr</code> if ownership sharing is not required	AUTOSAR A20-8-4	C++14	Rule
AUTOSAR C++14 Rule A20-8-5	<code>std::make_unique</code> shall be used to construct objects owned by <code>std::unique_ptr</code>	AUTOSAR A20-8-5	C++14	Rule
AUTOSAR C++14 Rule A20-8-6	<code>std::make_shared</code> shall be used to construct objects owned by <code>std::shared_ptr</code>	AUTOSAR A20-8-6	C++14	Rule
AUTOSAR C++14 Rule A20-8-7	A <code>std::weak_ptr</code> shall be used to represent temporary shared ownership.	AUTOSAR A20-8-7	C++14	Rule
AUTOSAR C++14 Rule A21-8-1	Arguments to character-handling functions shall be representable as an unsigned char	AUTOSAR A21-8-1	C++14	Rule
AUTOSAR C++14 Rule A23-0-1	An iterator shall not be implicitly converted to <code>const_iterator</code>	AUTOSAR A23-0-1	C++14	Rule
AUTOSAR C++14 Rule A23-0-2	Elements of a container shall only be accessed via valid references, iterators, and pointers	AUTOSAR A23-0-2	C++14	Rule
AUTOSAR C++14 Rule A25-1-1	Non-static data members or captured values of predicate function objects that are state related to this object's identity shall not be copied	AUTOSAR A25-1-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A25-4-1	Ordering predicates used with associative containers and STL sorting and related algorithms shall adhere to a strict weak ordering relation	AUTOSAR A25-4-1	C++14	Rule
AUTOSAR C++14 Rule A26-5-1	Pseudorandom numbers shall not be generated using <code>std::rand()</code>	AUTOSAR A26-5-1	C++14	Rule
AUTOSAR C++14 Rule A26-5-2	Random number engines shall not be default-initialized	AUTOSAR A26-5-2	C++14	Rule
AUTOSAR C++14 Rule A27-0-1	Inputs from independent components shall be validated.	AUTOSAR A27-0-1	C++14	Rule
AUTOSAR C++14 Rule A27-0-3	Alternate input and output operations on a file stream shall not be used without an intervening flush or positioning call	AUTOSAR A27-0-3	C++14	Rule
AUTOSAR C++14 Rule A27-0-4	C-style strings shall not be used	AUTOSAR A27-0-4	C++14	Rule
AUTOSAR C++14 Rule A3-1-1	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule	AUTOSAR A3-1-1	C++14	Rule
AUTOSAR C++14 Rule A3-1-2	Header files, that are defined locally in the project, shall have a file name extension of one of: <code>.h</code> , <code>.hpp</code> or <code>.hxx</code>	AUTOSAR A3-1-2	C++14	Rule
AUTOSAR C++14 Rule A3-1-4	When an array with external linkage is declared, its size shall be stated explicitly	AUTOSAR A3-1-4	C++14	Rule
AUTOSAR C++14 Rule A3-1-5	A function definition shall only be placed in a class definition if (1) the function is intended to be inlined (2) it is a member function template (3) it is a member function of a class template	AUTOSAR A3-1-5	C++14	Rule
AUTOSAR C++14 Rule A3-3-1	Objects or functions with external linkage (including members of named namespaces) shall be declared in a header file	AUTOSAR A3-3-1	C++14	Rule
AUTOSAR C++14 Rule A3-3-2	Static and thread-local objects shall be constant-initialized	AUTOSAR A3-3-2	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A3-8-1	An object shall not be accessed outside of its lifetime	AUTOSAR A3-8-1	C++14	Rule
AUTOSAR C++14 Rule A3-9-1	Fixed width integer types from <stdint>, indicating the size and signedness, shall be used in place of the basic numerical types	AUTOSAR A3-9-1	C++14	Rule
AUTOSAR C++14 Rule A4-10-1	Only nullptr literal shall be used as the null-pointer-constraint	AUTOSAR A4-10-1	C++14	Rule
AUTOSAR C++14 Rule A4-5-1	Expressions with type enum or enum class shall not be used as operands to built-in and overloaded operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=	AUTOSAR A4-5-1	C++14	Rule
AUTOSAR C++14 Rule A4-7-1	An integer expression shall not lead to data loss	AUTOSAR A4-7-1	C++14	Rule
AUTOSAR C++14 Rule A5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits	AUTOSAR A5-0-1	C++14	Rule
AUTOSAR C++14 Rule A5-0-2	The condition of an if-statement and the condition of an iteration statement shall have type bool	AUTOSAR A5-0-2	C++14	Rule
AUTOSAR C++14 Rule A5-0-3	The declaration of objects shall contain no more than two levels of pointer indirection	AUTOSAR A5-0-3	C++14	Rule
AUTOSAR C++14 Rule A5-0-4	Pointer arithmetic shall not be used with pointers to non-final classes	AUTOSAR A5-0-4	C++14	Rule
AUTOSAR C++14 Rule A5-1-1	Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead	AUTOSAR A5-1-1	C++14	Rule
AUTOSAR C++14 Rule A5-1-2	Variables shall not be implicitly captured in a lambda expression	AUTOSAR A5-1-2	C++14	Rule
AUTOSAR C++14 Rule A5-1-3	Parameter list (possibly empty) shall be included in every lambda expression	AUTOSAR A5-1-3	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A5-1-4	A lambda expression object shall not outlive any of its reference-captured objects	AUTOSAR A5-1-4	C++14	Rule
AUTOSAR C++14 Rule A5-1-7	A lambda shall not be an operand to decltype or typeid	AUTOSAR A5-1-7	C++14	Rule
AUTOSAR C++14 Rule A5-10-1	A pointer to member virtual function shall only be tested for equality with null-pointer-constant	AUTOSAR A5-10-1	C++14	Rule
AUTOSAR C++14 Rule A5-16-1	The ternary conditional operator shall not be used as a sub-expression	AUTOSAR A5-16-1	C++14	Rule
AUTOSAR C++14 Rule A5-2-2	Traditional C-style casts shall not be used	AUTOSAR A5-2-2	C++14	Rule
AUTOSAR C++14 Rule A5-2-3	A cast shall not remove any const or volatile qualification from the type of a pointer or reference	AUTOSAR A5-2-3	C++14	Rule
AUTOSAR C++14 Rule A5-2-4	reinterpret_cast shall not be used	AUTOSAR A5-2-4	C++14	Rule
AUTOSAR C++14 Rule A5-2-5	An array or container shall not be accessed beyond its range	AUTOSAR A5-2-5	C++14	Rule
AUTOSAR C++14 Rule A5-2-6	The operands of a logical && or shall be parenthesized if the operands contain binary operators	AUTOSAR A5-2-6	C++14	Rule
AUTOSAR C++14 Rule A5-3-1	Evaluation of the operand to the typeid operator shall not contain side effects	AUTOSAR A5-3-1	C++14	Rule
AUTOSAR C++14 Rule A5-3-2	Null pointers shall not be dereferenced	AUTOSAR A5-3-2	C++14	Rule
AUTOSAR C++14 Rule A5-3-3	Pointers to incomplete class types shall not be deleted	AUTOSAR A5-3-3	C++14	Rule
AUTOSAR C++14 Rule A5-5-1	A pointer to member shall not access non-existent class members	AUTOSAR A5-5-1	C++14	Rule
AUTOSAR C++14 Rule A5-6-1	The right hand operand of the integer division or remainder operators shall not be equal to zero	AUTOSAR A5-6-1	C++14	Rule
AUTOSAR C++14 Rule A6-2-1	Move and copy assignment operators shall either move or respectively copy base classes and data members of a class, without any side effects	AUTOSAR A6-2-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A6-2-2	Expression statements shall not be explicit calls to constructors of temporary objects only	AUTOSAR A6-2-2	C++14	Rule
AUTOSAR C++14 Rule A6-4-1	A switch statement shall have at least two case-clauses, distinct from the default label	AUTOSAR A6-4-1	C++14	Rule
AUTOSAR C++14 Rule A6-5-1	A for-loop that loops through all elements of the container and does not use its loop-counter shall not be used	AUTOSAR A6-5-1	C++14	Rule
AUTOSAR C++14 Rule A6-5-2	A for loop shall contain a single loop-counter which shall not have floating-point type	AUTOSAR A6-5-2	C++14	Rule
AUTOSAR C++14 Rule A6-6-1	The goto statement shall not be used	AUTOSAR A6-6-1	C++14	Rule
AUTOSAR C++14 Rule A7-1-1	Constexpr or const specifiers shall be used for immutable data declaration	AUTOSAR A7-1-1	C++14	Rule
AUTOSAR C++14 Rule A7-1-2	The constexpr specifier shall be used for values that can be determined at compile time	AUTOSAR A7-1-2	C++14	Rule
AUTOSAR C++14 Rule A7-1-3	CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name	AUTOSAR A7-1-3	C++14	Rule
AUTOSAR C++14 Rule A7-1-4	The register keyword shall not be used	AUTOSAR A7-1-4	C++14	Rule
AUTOSAR C++14 Rule A7-1-5	The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax	AUTOSAR A7-1-5	C++14	Rule
AUTOSAR C++14 Rule A7-1-6	The typedef specifier shall not be used	AUTOSAR A7-1-6	C++14	Rule
AUTOSAR C++14 Rule A7-1-7	Each expression statement and identifier declaration shall be placed on a separate line	AUTOSAR A7-1-7	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A7-1-8	A non-type specifier shall be placed before a type specifier in a declaration	AUTOSAR A7-1-8	C++14	Rule
AUTOSAR C++14 Rule A7-1-9	A class, structure, or enumeration shall not be declared in the definition of its type	AUTOSAR A7-1-9	C++14	Rule
AUTOSAR C++14 Rule A7-2-1	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration	AUTOSAR A7-2-1	C++14	Rule
AUTOSAR C++14 Rule A7-2-2	Enumeration underlying type shall be explicitly defined	AUTOSAR A7-2-2	C++14	Rule
AUTOSAR C++14 Rule A7-2-3	Enumerations shall be declared as scoped enum classes	AUTOSAR A7-2-3	C++14	Rule
AUTOSAR C++14 Rule A7-2-4	In an enumeration, either (1) none, (2) the first or (3) all enumerators shall be initialized	AUTOSAR A7-2-4	C++14	Rule
AUTOSAR C++14 Rule A7-3-1	All overloads of a function shall be visible from where it is called	AUTOSAR A7-3-1	C++14	Rule
AUTOSAR C++14 Rule A7-4-1	The asm declaration shall not be used	AUTOSAR A7-4-1	C++14	Rule
AUTOSAR C++14 Rule A7-5-1	A function shall not return a reference or a pointer to a parameter that is passed by reference to const	AUTOSAR A7-5-1	C++14	Rule
AUTOSAR C++14 Rule A7-5-2	Functions shall not call themselves, either directly or indirectly	AUTOSAR A7-5-2	C++14	Rule
AUTOSAR C++14 Rule A7-6-1	Functions declared with the [[noreturn]] attribute shall not return	AUTOSAR A7-6-1	C++14	Rule
AUTOSAR C++14 Rule A8-2-1	When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters	AUTOSAR A8-2-1	C++14	Rule
AUTOSAR C++14 Rule A8-4-1	Functions shall not be defined using the ellipsis notation	AUTOSAR A8-4-1	C++14	Rule
AUTOSAR C++14 Rule A8-4-10	A parameter shall be passed by reference if it can't be NULL	AUTOSAR A8-4-10	C++14	Rule
AUTOSAR C++14 Rule A8-4-11	A smart pointer shall only be used as a parameter type if it expresses lifetime semantics	AUTOSAR A8-4-11	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A8-4-12	A <code>std::unique_ptr</code> shall be passed to a function as: (1) a copy to express the function assumes ownership (2) an lvalue reference to express that the function replaces the managed object.	AUTOSAR A8-4-12	C++14	Rule
AUTOSAR C++14 Rule A8-4-13	A <code>std::shared_ptr</code> shall be passed to a function as: (1) a copy to express the function shares ownership (2) an lvalue reference to express that the function replaces the managed object (3) a <code>const</code> lvalue reference to express that the function retains a reference count.	AUTOSAR A8-4-13	C++14	Rule
AUTOSAR C++14 Rule A8-4-14	Interfaces shall be precisely and strongly typed	AUTOSAR A8-4-14	C++14	Rule
AUTOSAR C++14 Rule A8-4-2	All exit paths from a function with non-void return type shall have an explicit return statement with an expression	AUTOSAR A8-4-2	C++14	Rule
AUTOSAR C++14 Rule A8-4-5	"consume" parameters declared as <code>X &&</code> shall always be moved from	AUTOSAR A8-4-5	C++14	Rule
AUTOSAR C++14 Rule A8-4-6	"forward" parameters declared as <code>T &&</code> shall always be forwarded	AUTOSAR A8-4-6	C++14	Rule
AUTOSAR C++14 Rule A8-4-7	"in" parameters for "cheap to copy" types shall be passed by value	AUTOSAR A8-4-7	C++14	Rule
AUTOSAR C++14 Rule A8-4-8	Output parameters shall not be used	AUTOSAR A8-4-8	C++14	Rule
AUTOSAR C++14 Rule A8-4-9	"in-out" parameters declared as <code>T &</code> shall be modified	AUTOSAR A8-4-9	C++14	Rule
AUTOSAR C++14 Rule A8-5-0	All memory shall be initialized before it is read	AUTOSAR A8-5-0	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A8-5-1	In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition	AUTOSAR A8-5-1	C++14	Rule
AUTOSAR C++14 Rule A8-5-2	Braced-initialization {}, without equals sign, shall be used for variable initialization	AUTOSAR A8-5-2	C++14	Rule
AUTOSAR C++14 Rule A8-5-3	A variable of type auto shall not be initialized using {} or ={} braced-initialization	AUTOSAR A8-5-3	C++14	Rule
AUTOSAR C++14 Rule A9-3-1	Member functions shall not return non-constant "raw" pointers or references to private or protected data owned by the class	AUTOSAR A9-3-1	C++14	Rule
AUTOSAR C++14 Rule A9-5-1	Unions shall not be used	AUTOSAR A9-5-1	C++14	Rule
AUTOSAR C++14 Rule A9-6-1	Data types used for interfacing with hardware or conforming to communication protocols shall be trivial, standard-layout and only contain members of types with defined sizes	AUTOSAR A9-6-1	C++14	Rule
AUTOSAR C++14 Rule M0-1-1	A project shall not contain unreachable code	AUTOSAR M0-1-1	C++14	Rule
AUTOSAR C++14 Rule M0-1-2	A project shall not contain infeasible paths	AUTOSAR M0-1-2	C++14	Rule
AUTOSAR C++14 Rule M0-1-3	A project shall not contain unused variables	AUTOSAR M0-1-3	C++14	Rule
AUTOSAR C++14 Rule M0-1-4	A project shall not contain non-volatile POD variables having only one use	AUTOSAR M0-1-4	C++14	Rule
AUTOSAR C++14 Rule M0-1-8	All functions with void return type shall have external side effect(s)	AUTOSAR M0-1-8	C++14	Rule
AUTOSAR C++14 Rule M0-1-9	There shall be no dead code	AUTOSAR M0-1-9	C++14	Rule
AUTOSAR C++14 Rule M0-2-1	An object shall not be assigned to an overlapping object	AUTOSAR M0-2-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M0-3-2	If a function generates error information, then that error information shall be tested	AUTOSAR M0-3-2	C++14	Rule
AUTOSAR C++14 Rule M10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy	AUTOSAR M10-1-2	C++14	Rule
AUTOSAR C++14 Rule M10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy	AUTOSAR M10-1-3	C++14	Rule
AUTOSAR C++14 Rule M10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual	AUTOSAR M10-3-3	C++14	Rule
AUTOSAR C++14 Rule M11-0-1	Member data in non-POD class types shall be private	AUTOSAR M11-0-1	C++14	Rule
AUTOSAR C++14 Rule M12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor	AUTOSAR M12-1-1	C++14	Rule
AUTOSAR C++14 Rule M14-5-3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter	AUTOSAR M14-5-3	C++14	Rule
AUTOSAR C++14 Rule M14-6-1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->	AUTOSAR M14-6-1	C++14	Rule
AUTOSAR C++14 Rule M15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement	AUTOSAR M15-0-3	C++14	Rule
AUTOSAR C++14 Rule M15-1-1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown	AUTOSAR M15-1-1	C++14	Rule
AUTOSAR C++14 Rule M15-1-2	NULL shall not be thrown explicitly	AUTOSAR M15-1-2	C++14	Rule
AUTOSAR C++14 Rule M15-1-3	An empty throw (throw;) shall only be used in the compound statement of a catch handler	AUTOSAR M15-1-3	C++14	Rule
AUTOSAR C++14 Rule M15-3-1	Exceptions shall be raised only after start-up and before termination	AUTOSAR M15-3-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases	AUTOSAR M15-3-3	C++14	Rule
AUTOSAR C++14 Rule M15-3-4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point	AUTOSAR M15-3-4	C++14	Rule
AUTOSAR C++14 Rule M15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class	AUTOSAR M15-3-6	C++14	Rule
AUTOSAR C++14 Rule M15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last	AUTOSAR M15-3-7	C++14	Rule
AUTOSAR C++14 Rule M16-0-1	#include directives in a file shall only be preceded by other preprocessor directives or comments	AUTOSAR M16-0-1	C++14	Rule
AUTOSAR C++14 Rule M16-0-2	Macros shall only be #define'd or #undef'd in the global namespace	AUTOSAR M16-0-2	C++14	Rule
AUTOSAR C++14 Rule M16-0-5	Arguments to a function-like macro shall not contain tokens that look like pre-processing directives	AUTOSAR M16-0-5	C++14	Rule
AUTOSAR C++14 Rule M16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##	AUTOSAR M16-0-6	C++14	Rule
AUTOSAR C++14 Rule M16-0-7	Undefined macro identifiers shall not be used in #if or #elif pre-processor directives, except as operands to the defined operator	AUTOSAR M16-0-7	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M16-0-8	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token	AUTOSAR M16-0-8	C++14	Rule
AUTOSAR C++14 Rule M16-1-1	The defined pre-processor operator shall only be used in one of the two standard forms	AUTOSAR M16-1-1	C++14	Rule
AUTOSAR C++14 Rule M16-1-2	All #else, #elif and #endif pre-processor directives shall reside in the same file as the #if or #ifdef directive to which they are related	AUTOSAR M16-1-2	C++14	Rule
AUTOSAR C++14 Rule M16-2-3	Include guards shall be provided	AUTOSAR M16-2-3	C++14	Rule
AUTOSAR C++14 Rule M16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition	AUTOSAR M16-3-1	C++14	Rule
AUTOSAR C++14 Rule M17-0-2	The names of standard library macros and objects shall not be reused	AUTOSAR M17-0-2	C++14	Rule
AUTOSAR C++14 Rule M17-0-3	The names of standard library functions shall not be overridden	AUTOSAR M17-0-3	C++14	Rule
AUTOSAR C++14 Rule M17-0-5	The setjmp macro and the longjmp function shall not be used	AUTOSAR M17-0-5	C++14	Rule
AUTOSAR C++14 Rule M18-0-3	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used	AUTOSAR M18-0-3	C++14	Rule
AUTOSAR C++14 Rule M18-0-4	The time handling functions of library <ctime> shall not be used	AUTOSAR M18-0-4	C++14	Rule
AUTOSAR C++14 Rule M18-0-5	The unbounded functions of library <cstring> shall not be used	AUTOSAR M18-0-5	C++14	Rule
AUTOSAR C++14 Rule M18-2-1	The macro offsetof shall not be used	AUTOSAR M18-2-1	C++14	Rule
AUTOSAR C++14 Rule M18-7-1	The signal handling facilities of <csignal> shall not be used	AUTOSAR M18-7-1	C++14	Rule
AUTOSAR C++14 Rule M19-3-1	The error indicator errno shall not be used	AUTOSAR M19-3-1	C++14	Rule
AUTOSAR C++14 Rule M2-10-1	Different identifiers shall be typographically unambiguous	AUTOSAR M2-10-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M2-13-2	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used	AUTOSAR M2-13-2	C++14	Rule
AUTOSAR C++14 Rule M2-13-3	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type	AUTOSAR M2-13-3	C++14	Rule
AUTOSAR C++14 Rule M2-13-4	Literal suffixes shall be upper case	AUTOSAR M2-13-4	C++14	Rule
AUTOSAR C++14 Rule M2-7-1	The character sequence /* shall not be used within a C-style comment	AUTOSAR M2-7-1	C++14	Rule
AUTOSAR C++14 Rule M27-0-1	The stream input/output library <cstdio> shall not be used	AUTOSAR M27-0-1	C++14	Rule
AUTOSAR C++14 Rule M3-1-2	Functions shall not be declared at block scope	AUTOSAR M3-1-2	C++14	Rule
AUTOSAR C++14 Rule M3-2-1	All declarations of an object or function shall have compatible types	AUTOSAR M3-2-1	C++14	Rule
AUTOSAR C++14 Rule M3-2-2	The One Definition Rule shall not be violated	AUTOSAR M3-2-2	C++14	Rule
AUTOSAR C++14 Rule M3-2-3	A type, object or function that is used in multiple translation units shall be declared in one and only one file	AUTOSAR M3-2-3	C++14	Rule
AUTOSAR C++14 Rule M3-2-4	An identifier with external linkage shall have exactly one definition	AUTOSAR M3-2-4	C++14	Rule
AUTOSAR C++14 Rule M3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier	AUTOSAR M3-3-2	C++14	Rule
AUTOSAR C++14 Rule M3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility	AUTOSAR M3-4-1	C++14	Rule
AUTOSAR C++14 Rule M3-9-1	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations	AUTOSAR M3-9-1	C++14	Rule
AUTOSAR C++14 Rule M3-9-3	The underlying bit representations of floating-point values shall not be used	AUTOSAR M3-9-3	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M4-10-1	NULL shall not be used as an integer value	AUTOSAR M4-10-1	C++14	Rule
AUTOSAR C++14 Rule M4-10-2	Literal zero (0) shall not be used as the null-pointer-constant	AUTOSAR M4-10-2	C++14	Rule
AUTOSAR C++14 Rule M4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, , !, the equality operators == and !=, the unary & operator, and the conditional operator	AUTOSAR M4-5-1	C++14	Rule
AUTOSAR C++14 Rule M4-5-3	Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator	AUTOSAR M4-5-3	C++14	Rule
AUTOSAR C++14 Rule M5-0-10	If the bitwise operators ~and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand	AUTOSAR M5-0-10	C++14	Rule
AUTOSAR C++14 Rule M5-0-11	The plain char type shall only be used for the storage and use of character values	AUTOSAR M5-0-11	C++14	Rule
AUTOSAR C++14 Rule M5-0-12	Signed char and unsigned char type shall only be used for the storage and use of numeric values	AUTOSAR M5-0-12	C++14	Rule
AUTOSAR C++14 Rule M5-0-14	The first operand of a conditional-operator shall have type bool	AUTOSAR M5-0-14	C++14	Rule
AUTOSAR C++14 Rule M5-0-15	Array indexing shall be the only form of pointer arithmetic	AUTOSAR M5-0-15	C++14	Rule
AUTOSAR C++14 Rule M5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array	AUTOSAR M5-0-16	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M5-0-17	Subtraction between pointers shall only be applied to pointers that address elements of the same array	AUTOSAR M5-0-17	C++14	Rule
AUTOSAR C++14 Rule M5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array	AUTOSAR M5-0-18	C++14	Rule
AUTOSAR C++14 Rule M5-0-20	Non-constant operands to a binary bitwise operator shall have the same underlying type	AUTOSAR M5-0-20	C++14	Rule
AUTOSAR C++14 Rule M5-0-21	Bitwise operators shall only be applied to operands of unsigned underlying type	AUTOSAR M5-0-21	C++14	Rule
AUTOSAR C++14 Rule M5-0-3	A cvalue expression shall not be implicitly converted to a different underlying type	AUTOSAR M5-0-3	C++14	Rule
AUTOSAR C++14 Rule M5-0-4	An implicit integral conversion shall not change the signedness of the underlying type	AUTOSAR M5-0-4	C++14	Rule
AUTOSAR C++14 Rule M5-0-5	There shall be no implicit floating-integral conversions	AUTOSAR M5-0-5	C++14	Rule
AUTOSAR C++14 Rule M5-0-6	An implicit integral or floating-point conversion shall not reduce the size of the underlying type	AUTOSAR M5-0-6	C++14	Rule
AUTOSAR C++14 Rule M5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression	AUTOSAR M5-0-7	C++14	Rule
AUTOSAR C++14 Rule M5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression	AUTOSAR M5-0-8	C++14	Rule
AUTOSAR C++14 Rule M5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression	AUTOSAR M5-0-9	C++14	Rule
AUTOSAR C++14 Rule M5-14-1	The right hand operand of a logical &&, operators shall not contain side effects	AUTOSAR M5-14-1	C++14	Rule
AUTOSAR C++14 Rule M5-18-1	The comma operator shall not be used	AUTOSAR M5-18-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M5-19-1	Evaluation of constant unsigned integer expressions shall not lead to wrap-around	AUTOSAR M5-19-1	C++14	Rule
AUTOSAR C++14 Rule M5-2-10	The increment (++) and decrement (--) operators shall not be mixed with other operators in an expression	AUTOSAR M5-2-10	C++14	Rule
AUTOSAR C++14 Rule M5-2-11	The comma operator, && operator and the operator shall not be overloaded	AUTOSAR M5-2-11	C++14	Rule
AUTOSAR C++14 Rule M5-2-12	An identifier with array type passed as a function argument shall not decay to a pointer	AUTOSAR M5-2-12	C++14	Rule
AUTOSAR C++14 Rule M5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of <code>dynamic_cast</code>	AUTOSAR M5-2-2	C++14	Rule
AUTOSAR C++14 Rule M5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type	AUTOSAR M5-2-6	C++14	Rule
AUTOSAR C++14 Rule M5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type	AUTOSAR M5-2-8	C++14	Rule
AUTOSAR C++14 Rule M5-2-9	A cast shall not convert a pointer type to an integral type	AUTOSAR M5-2-9	C++14	Rule
AUTOSAR C++14 Rule M5-3-1	Each operand of the ! operator, the logical && or the logical operators shall have type bool	AUTOSAR M5-3-1	C++14	Rule
AUTOSAR C++14 Rule M5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned	AUTOSAR M5-3-2	C++14	Rule
AUTOSAR C++14 Rule M5-3-3	The unary & operator shall not be overloaded	AUTOSAR M5-3-3	C++14	Rule
AUTOSAR C++14 Rule M5-3-4	Evaluation of the operand to the sizeof operator shall not contain side effects	AUTOSAR M5-3-4	C++14	Rule
AUTOSAR C++14 Rule M5-8-1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand	AUTOSAR M5-8-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M6-2-1	Assignment operators shall not be used in sub-expressions	AUTOSAR M6-2-1	C++14	Rule
AUTOSAR C++14 Rule M6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality	AUTOSAR M6-2-2	C++14	Rule
AUTOSAR C++14 Rule M6-2-3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character	AUTOSAR M6-2-3	C++14	Rule
AUTOSAR C++14 Rule M6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement	AUTOSAR M6-3-1	C++14	Rule
AUTOSAR C++14 Rule M6-4-1	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement	AUTOSAR M6-4-1	C++14	Rule
AUTOSAR C++14 Rule M6-4-2	All if ... else if constructs shall be terminated with an else clause	AUTOSAR M6-4-2	C++14	Rule
AUTOSAR C++14 Rule M6-4-3	A switch statement shall be a well-formed switch statement	AUTOSAR M6-4-3	C++14	Rule
AUTOSAR C++14 Rule M6-4-4	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	AUTOSAR M6-4-4	C++14	Rule
AUTOSAR C++14 Rule M6-4-5	An unconditional throw or break statement shall terminate every non-empty switch-clause	AUTOSAR M6-4-5	C++14	Rule
AUTOSAR C++14 Rule M6-4-6	The final clause of a switch statement shall be the default-clause	AUTOSAR M6-4-6	C++14	Rule
AUTOSAR C++14 Rule M6-4-7	The condition of a switch statement shall not have bool type	AUTOSAR M6-4-7	C++14	Rule
AUTOSAR C++14 Rule M6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=	AUTOSAR M6-5-2	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M6-5-3	The loop-counter shall not be modified within condition or statement	AUTOSAR M6-5-3	C++14	Rule
AUTOSAR C++14 Rule M6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop	AUTOSAR M6-5-4	C++14	Rule
AUTOSAR C++14 Rule M6-5-5	A loop-control-variable other than the loop-counter shall not be modified within condition or expression	AUTOSAR M6-5-5	C++14	Rule
AUTOSAR C++14 Rule M6-5-6	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool	AUTOSAR M6-5-6	C++14	Rule
AUTOSAR C++14 Rule M6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement	AUTOSAR M6-6-1	C++14	Rule
AUTOSAR C++14 Rule M6-6-2	The goto statement shall jump to a label declared later in the same function body	AUTOSAR M6-6-2	C++14	Rule
AUTOSAR C++14 Rule M6-6-3	The continue statement shall only be used within a well-formed for loop	AUTOSAR M6-6-3	C++14	Rule
AUTOSAR C++14 Rule M7-1-2	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified	AUTOSAR M7-1-2	C++14	Rule
AUTOSAR C++14 Rule M7-3-1	The global namespace shall only contain main, namespace declarations and extern "C" declarations	AUTOSAR M7-3-1	C++14	Rule
AUTOSAR C++14 Rule M7-3-2	The identifier main shall not be used for a function other than the global function main	AUTOSAR M7-3-2	C++14	Rule
AUTOSAR C++14 Rule M7-3-3	There shall be no unnamed namespaces in header files	AUTOSAR M7-3-3	C++14	Rule
AUTOSAR C++14 Rule M7-3-4	Using-directives shall not be used	AUTOSAR M7-3-4	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M7-3-6	Using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files	AUTOSAR M7-3-6	C++14	Rule
AUTOSAR C++14 Rule M7-4-2	Assembler instructions shall only be introduced using the asm declaration	AUTOSAR M7-4-2	C++14	Rule
AUTOSAR C++14 Rule M7-4-3	Assembly language shall be encapsulated and isolated	AUTOSAR M7-4-3	C++14	Rule
AUTOSAR C++14 Rule M7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function	AUTOSAR M7-5-1	C++14	Rule
AUTOSAR C++14 Rule M7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist	AUTOSAR M7-5-2	C++14	Rule
AUTOSAR C++14 Rule M8-0-1	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively	AUTOSAR M8-0-1	C++14	Rule
AUTOSAR C++14 Rule M8-3-1	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments	AUTOSAR M8-3-1	C++14	Rule
AUTOSAR C++14 Rule M8-4-2	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration	AUTOSAR M8-4-2	C++14	Rule
AUTOSAR C++14 Rule M8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &	AUTOSAR M8-4-4	C++14	Rule
AUTOSAR C++14 Rule M8-5-2	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures	AUTOSAR M8-5-2	C++14	Rule
AUTOSAR C++14 Rule M9-3-1	Const member functions shall not return non-const pointers or references to class-data	AUTOSAR M9-3-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M9-3-3	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const	AUTOSAR C++14 Rule M9-3-3
AUTOSAR C++14 Rule M9-6-4	Named bit-fields with signed integer type shall have a length of more than one bit	AUTOSAR C++14 Rule M9-6-4

Unsupported Rules

Polyspace does not support these **Required** rules:

Rule	Description
M0-3-1	Minimization of run-time failures shall be ensured by the use of at least one of:\n (a) static analysis tools/techniques;\n (b) dynamic analysis tools/techniques;\n (c) explicit coding of checks to handle run-time faults.
M0-4-1	Use of scaled-integer or fixed-point arithmetic shall be documented.
M0-4-2	Use of floating-point arithmetic shall be documented.
A0-4-1	Floating-point implementation shall comply with IEEE 754 standard.
A0-4-3	The implementations in the chosen compiler shall strictly comply with the C++14 Language Standard.
M1-0-2	Multiple compilers shall only be used if they have a common, defined interface.
A1-1-2	A warning level of the compilation process shall be set in compliance with project policies.
A1-1-3	An optimization option that disregards strict standard compliance shall not be turned on in the chosen compiler.
A1-2-1	When using a compiler toolchain (including preprocessor, compiler itself, linker, C++ standard libraries) in safety-related software, the tool confidence level (TCL) shall be determined. In case of TCL2 or TCL3, the compiler shall undergo a "Qualification of a software tool", as per ISO 26262-8.11.4.6 [5].
A1-4-1	Code metrics and their valid boundaries shall be defined and code shall comply with defined boundaries of code metrics.

Rule	Description
A2-7-5	Comments shall not document any actions or sources (e.g. tables, figures, paragraphs, etc.) that are outside of the file.
M5-17-1	The semantic equivalence between a binary operator and its assignment operator form shall be preserved.
M7-4-1	All usage of assembler shall be documented.
M9-6-1	When the absolute positioning of bits representing a bit-field is required, then the behavior and packing of bit-fields shall be documented.
A9-6-2	Bit-fields shall be used only when interfacing to hardware or conforming to communication protocols.
A10-0-1	Public inheritance shall be used to implement "is-a" relationship.
A10-0-2	Membership or non-public inheritance shall be used to implement "has-a" relationship.
A15-0-1	A function shall not exit with an exception if it is able to complete its task.
A15-0-4	Unchecked exceptions shall be used to represent errors from which the caller cannot reasonably be expected to recover.
A15-0-5	Checked exceptions shall be used to represent errors from which the caller can reasonably be expected to recover.
A15-0-6	An analysis shall be performed to analyze the failure modes of exception handling. In particular, the following failure modes shall be analyzed:\n (a) worst time execution time not existing or cannot be determined,\n (b) stack not correctly unwound,\n (c) exception not thrown, other exception thrown, wrong catch activated,\n (d) memory not available while exception handling.
A15-0-8	A worst-case execution time (WCET) analysis shall be performed to determine maximum execution time constraints of the software, covering in particular the exceptions processing.
A15-3-2	If a function throws an exception, it shall be handled when meaningful actions can be taken, otherwise it shall be propagated.
A17-0-2	All project's code including used libraries (including standard and user-defined libraries) and any third-party user code shall conform to the AUTOSAR C++14 Coding Guidelines.

Rule	Description
A18-5-6	An analysis shall be performed to analyze the failure modes of dynamic memory management. In particular, the following failure modes shall be analyzed:\n (a) non-deterministic behavior resulting with nonexistence of worst-case execution time,\n (b) memory fragmentation,\n (c) running out of memory,\n (d) mismatched allocations and deallocations,\n (e) dependence on non-deterministic calls to kernel.

See Also

Check AUTOSAR C++ 14 (-autosar-cpp14)

More About

- “Check for and Review Coding Standard Violations” on page 16-2
- “Coding Standards”
- “Statically Enforceable AUTOSAR C++14 Rules Supported by Polyspace Bug Finder” on page 17-156
- “Required MISRA C++:2008 Coding Rules Supported by Polyspace Bug Finder” on page 17-83
- “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78

Statically Enforceable AUTOSAR C++14 Rules Supported by Polyspace Bug Finder

The AUTOSAR C++14 standard classifies the rules that are statically enforceable as **Automated** and **Partially Automated**. In total, Polyspace supports 349 out of 349² AUTOSAR C++14 coding rules that are enforceable by a static analysis tool.

Automated Rules

According to the AUTOSAR C++14 standard, static analysis detects all violations of the **Automated** rules. Polyspace Bug Finder supports 327 out of 327 **Automated** rules that can be enforced by a static analysis tool. The AUTOSAR C++14 standard contains two **Automated** rules that cannot be enforced by a static analysis tool.

Polyspace supports these **Automated** rules.

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A0-1-1	A project shall not contain instances of non-volatile variables being given values that are not subsequently used	AUTOSAR A0-1-1	C++14	Rule
AUTOSAR C++14 Rule A0-1-2	The value returned by a function having a non-void return type that is not an overloaded operator shall be used	AUTOSAR A0-1-2	C++14	Rule
AUTOSAR C++14 Rule A0-1-3	Every function defined in an anonymous namespace, or static function with internal linkage, or private member function shall be used	AUTOSAR A0-1-3	C++14	Rule
AUTOSAR C++14 Rule A0-1-4	There shall be no unused named parameters in non-virtual functions	AUTOSAR A0-1-4	C++14	Rule
AUTOSAR C++14 Rule A0-1-5	There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it	AUTOSAR A0-1-5	C++14	Rule
AUTOSAR C++14 Rule A0-1-6	There should be no unused type declarations	AUTOSAR A0-1-6	C++14	Rule
AUTOSAR C++14 Rule A0-4-2	Type long double shall not be used	AUTOSAR A0-4-2	C++14	Rule

² The AUTOSAR C++14 standard contains 351 statically enforceable rules. The rules A0-4-3 and A1-4-3 are not enforceable by a static analysis tool. These rules might be enforced by your compiler.

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A1-1-1	All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features	AUTOSAR A1-1-1	C++14	Rule
AUTOSAR C++14 Rule A10-1-1	Class shall not be derived from more than one base class which is not an interface class	AUTOSAR A10-1-1	C++14	Rule
AUTOSAR C++14 Rule A10-2-1	Non-virtual public or protected member functions shall not be redefined in derived classes	AUTOSAR A10-2-1	C++14	Rule
AUTOSAR C++14 Rule A10-3-1	Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final	AUTOSAR A10-3-1	C++14	Rule
AUTOSAR C++14 Rule A10-3-2	Each overriding virtual function shall be declared with the override or final specifier	AUTOSAR A10-3-2	C++14	Rule
AUTOSAR C++14 Rule A10-3-3	Virtual functions shall not be introduced in a final class	AUTOSAR A10-3-3	C++14	Rule
AUTOSAR C++14 Rule A10-3-5	A user-defined assignment operator shall not be virtual	AUTOSAR A10-3-5	C++14	Rule
AUTOSAR C++14 Rule A11-0-1	A non-POD type should be defined as class	AUTOSAR A11-0-1	C++14	Rule
AUTOSAR C++14 Rule A11-0-2	A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class	AUTOSAR A11-0-2	C++14	Rule
AUTOSAR C++14 Rule A11-3-1	Friend declarations shall not be used	AUTOSAR A11-3-1	C++14	Rule
AUTOSAR C++14 Rule A12-0-1	If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well	AUTOSAR A12-0-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A12-1-1	Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members	AUTOSAR A12-1-1	C++14	Rule
AUTOSAR C++14 Rule A12-1-2	Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type	AUTOSAR A12-1-2	C++14	Rule
AUTOSAR C++14 Rule A12-1-3	If all user-defined constructors of a class initialize data members with constant values that are the same across all constructors, then data members shall be initialized using NSDMI instead	AUTOSAR A12-1-3	C++14	Rule
AUTOSAR C++14 Rule A12-1-4	All constructors that are callable with a single argument of fundamental type shall be declared explicit	AUTOSAR A12-1-4	C++14	Rule
AUTOSAR C++14 Rule A12-1-6	Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors	AUTOSAR A12-1-6	C++14	Rule
AUTOSAR C++14 Rule A12-4-1	Destructor of a base class shall be public virtual, public override or protected non-virtual	AUTOSAR A12-4-1	C++14	Rule
AUTOSAR C++14 Rule A12-4-2	If a public destructor of a class is non-virtual, then the class should be declared final	AUTOSAR A12-4-2	C++14	Rule
AUTOSAR C++14 Rule A12-6-1	All class data members that are initialized by the constructor shall be initialized using member initializers	AUTOSAR A12-6-1	C++14	Rule
AUTOSAR C++14 Rule A12-7-1	If the behavior of a user-defined special member function is identical to implicitly defined special member function, then it shall be defined "=default" or be left undefined	AUTOSAR A12-7-1	C++14	Rule
AUTOSAR C++14 Rule A12-8-1	Move and copy constructors shall move and respectively copy base classes and data members of a class, without any side effects	AUTOSAR A12-8-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A12-8-2	User-defined copy and move assignment operators should use user-defined no-throw swap function	AUTOSAR A12-8-2	C++14	Rule
AUTOSAR C++14 Rule A12-8-4	Move constructor shall not initialize its class members and base classes using copy semantics	AUTOSAR A12-8-4	C++14	Rule
AUTOSAR C++14 Rule A12-8-5	A copy assignment and a move assignment operators shall handle self-assignment	AUTOSAR A12-8-5	C++14	Rule
AUTOSAR C++14 Rule A12-8-6	Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class	AUTOSAR A12-8-6	C++14	Rule
AUTOSAR C++14 Rule A12-8-7	Assignment operators should be declared with the ref-qualifier &	AUTOSAR A12-8-7	C++14	Rule
AUTOSAR C++14 Rule A13-1-2	User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters	AUTOSAR A13-1-2	C++14	Rule
AUTOSAR C++14 Rule A13-1-3	User defined literals operators shall only perform conversion of passed parameters	AUTOSAR A13-1-3	C++14	Rule
AUTOSAR C++14 Rule A13-2-1	An assignment operator shall return a reference to "this"	AUTOSAR A13-2-1	C++14	Rule
AUTOSAR C++14 Rule A13-2-2	A binary arithmetic operator and a bitwise operator shall return a "prvalue"	AUTOSAR A13-2-2	C++14	Rule
AUTOSAR C++14 Rule A13-2-3	A relational operator shall return a boolean value	AUTOSAR A13-2-3	C++14	Rule
AUTOSAR C++14 Rule A13-3-1	A function that contains "forwarding reference" as its argument shall not be overloaded	AUTOSAR A13-3-1	C++14	Rule
AUTOSAR C++14 Rule A13-5-1	If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented	AUTOSAR A13-5-1	C++14	Rule
AUTOSAR C++14 Rule A13-5-2	All user-defined conversion operators shall be defined explicit	AUTOSAR A13-5-2	C++14	Rule
AUTOSAR C++14 Rule A13-5-3	User-defined conversion operators should not be used	AUTOSAR A13-5-3	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A13-5-4	If two opposite operators are defined, one shall be defined in terms of the other	AUTOSAR A13-5-4	C++14	Rule
AUTOSAR C++14 Rule A13-5-5	Comparison operators shall be non-member functions with identical parameter types and noexcept	AUTOSAR A13-5-5	C++14	Rule
AUTOSAR C++14 Rule A13-6-1	Digit sequences separators ' shall only be used as follows: (1) for decimal, every 3 digits, (2) for hexadecimal, every 2 digits, (3) for binary, every 4 digits	AUTOSAR A13-6-1	C++14	Rule
AUTOSAR C++14 Rule A14-5-1	A template constructor shall not participate in overload resolution for a single argument of the enclosing class type	AUTOSAR A14-5-1	C++14	Rule
AUTOSAR C++14 Rule A14-5-3	A non-member generic operator shall only be declared in a namespace that does not contain class (struct) type, enum type or union type declarations	AUTOSAR A14-5-3	C++14	Rule
AUTOSAR C++14 Rule A14-7-1	A type used as a template argument shall provide all members that are used by the template	AUTOSAR A14-7-1	C++14	Rule
AUTOSAR C++14 Rule A14-7-2	Template specialization shall be declared in the same file (1) as the primary template (2) as a user-defined type, for which the specialization is declared	AUTOSAR A14-7-2	C++14	Rule
AUTOSAR C++14 Rule A14-8-2	Explicit specializations of function templates shall not be used	AUTOSAR A14-8-2	C++14	Rule
AUTOSAR C++14 Rule A15-1-1	Only instances of types derived from std::exception should be thrown	AUTOSAR A15-1-1	C++14	Rule
AUTOSAR C++14 Rule A15-1-2	An exception object shall not be a pointer	AUTOSAR A15-1-2	C++14	Rule
AUTOSAR C++14 Rule A15-1-3	All thrown exceptions should be unique	AUTOSAR A15-1-3	C++14	Rule
AUTOSAR C++14 Rule A15-2-1	Constructors that are not noexcept shall not be invoked before program startup	AUTOSAR A15-2-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A15-3-5	A class type exception shall be caught by reference or const reference	AUTOSAR A15-3-5	C++14	Rule
AUTOSAR C++14 Rule A15-4-1	Dynamic exception-specification shall not be used	AUTOSAR A15-4-1	C++14	Rule
AUTOSAR C++14 Rule A15-4-2	If a function is declared to be <code>noexcept</code> , <code>noexcept(true)</code> or <code>noexcept(<true condition>)</code> , then it shall not exit with an exception	AUTOSAR A15-4-2	C++14	Rule
AUTOSAR C++14 Rule A15-4-3	The <code>noexcept</code> specification of a function shall either be identical across all translation units, or identical or more restrictive between a virtual member function and an overrider	AUTOSAR A15-4-3	C++14	Rule
AUTOSAR C++14 Rule A15-4-4	A declaration of non-throwing function shall contain <code>noexcept</code> specification	AUTOSAR A15-4-4	C++14	Rule
AUTOSAR C++14 Rule A15-4-5	Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overriders	AUTOSAR A15-4-5	C++14	Rule
AUTOSAR C++14 Rule A15-5-1	All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A <code>noexcept</code> exception specification shall be added to these functions as appropriate	AUTOSAR A15-5-1	C++14	Rule
AUTOSAR C++14 Rule A15-5-3	The <code>std::terminate()</code> function shall not be called implicitly	AUTOSAR A15-5-3	C++14	Rule
AUTOSAR C++14 Rule A16-0-1	The preprocessor shall only be used for unconditional and conditional file inclusion and include guards, and using specific directives	AUTOSAR A16-0-1	C++14	Rule
AUTOSAR C++14 Rule A16-2-1	The <code>'</code> , <code>"</code> , <code>/*</code> , <code>//</code> , <code>\</code> characters shall not occur in a header file name or in <code>#include</code> directive	AUTOSAR A16-2-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A16-2-2	There shall be no unused include directives	AUTOSAR A16-2-2	C++14	Rule
AUTOSAR C++14 Rule A16-6-1	#error directive shall not be used	AUTOSAR A16-6-1	C++14	Rule
AUTOSAR C++14 Rule A16-7-1	The #pragma directive shall not be used	AUTOSAR A16-7-1	C++14	Rule
AUTOSAR C++14 Rule A17-0-1	Reserved identifiers, macros and functions in the C++ standard library shall not be defined, redefined or undefined	AUTOSAR A17-0-1	C++14	Rule
AUTOSAR C++14 Rule A17-6-1	Non-standard entities shall not be added to standard namespaces	AUTOSAR A17-6-1	C++14	Rule
AUTOSAR C++14 Rule A18-0-1	The C library facilities shall only be accessed through C++ library headers	AUTOSAR A18-0-1	C++14	Rule
AUTOSAR C++14 Rule A18-0-2	The error state of a conversion from string to a numeric value shall be checked	AUTOSAR A18-0-2	C++14	Rule
AUTOSAR C++14 Rule A18-0-3	The library <locale> (locale.h) and the setlocale function shall not be used	AUTOSAR A18-0-3	C++14	Rule
AUTOSAR C++14 Rule A18-1-1	C-style arrays shall not be used	AUTOSAR A18-1-1	C++14	Rule
AUTOSAR C++14 Rule A18-1-2	The std::vector<bool> specialization shall not be used	AUTOSAR A18-1-2	C++14	Rule
AUTOSAR C++14 Rule A18-1-3	The std::auto_ptr shall not be used	AUTOSAR A18-1-3	C++14	Rule
AUTOSAR C++14 Rule A18-1-4	A pointer pointing to an element of an array of objects shall not be passed to a smart pointer of single object type	AUTOSAR A18-1-4	C++14	Rule
AUTOSAR C++14 Rule A18-1-6	All std::hash specializations for user-defined types shall have a noexcept function call operator	AUTOSAR A18-1-6	C++14	Rule
AUTOSAR C++14 Rule A18-5-1	Functions malloc, calloc, realloc and free shall not be used	AUTOSAR A18-5-1	C++14	Rule
AUTOSAR C++14 Rule A18-5-10	Placement new shall be used only with properly aligned pointers to sufficient storage capacity	AUTOSAR A18-5-10	C++14	Rule
AUTOSAR C++14 Rule A18-5-11	"operator new" and "operator delete" shall be defined together	AUTOSAR A18-5-11	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A18-5-3	The form of delete operator shall match the form of new operator used to allocate the memory	AUTOSAR A18-5-3	C++14	Rule
AUTOSAR C++14 Rule A18-5-4	If a project has sized or unsized version of operator 'delete' globally defined, then both sized and unsized versions shall be defined	AUTOSAR A18-5-4	C++14	Rule
AUTOSAR C++14 Rule A18-5-9	Custom implementations of dynamic memory allocation and deallocation functions shall meet the semantic requirements specified in the corresponding "Required behaviour" clause from the C++ Standard	AUTOSAR A18-5-9	C++14	Rule
AUTOSAR C++14 Rule A18-9-1	The std::bind shall not be used	AUTOSAR A18-9-1	C++14	Rule
AUTOSAR C++14 Rule A18-9-2	Forwarding values to other functions shall be done via: (1) std::move if the value is an rvalue reference, (2) std::forward if the value is forwarding reference	AUTOSAR A18-9-2	C++14	Rule
AUTOSAR C++14 Rule A18-9-3	The std::move shall not be used on objects declared const or const&	AUTOSAR A18-9-3	C++14	Rule
AUTOSAR C++14 Rule A18-9-4	An argument to std::forward shall not be subsequently used	AUTOSAR A18-9-4	C++14	Rule
AUTOSAR C++14 Rule A2-10-1	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope	AUTOSAR A2-10-1	C++14	Rule
AUTOSAR C++14 Rule A2-10-4	The identifier name of a non-member object with static storage duration or static function shall not be reused within a namespace	AUTOSAR A2-10-4	C++14	Rule
AUTOSAR C++14 Rule A2-10-5	An identifier name of a function with static storage duration or a non-member object with external or internal linkage should not be reused	AUTOSAR A2-10-5	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A2-10-6	A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope	AUTOSAR A2-10-6	C++14	Rule
AUTOSAR C++14 Rule A2-11-1	Volatile keyword shall not be used	AUTOSAR A2-11-1	C++14	Rule
AUTOSAR C++14 Rule A2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2014 shall be used	AUTOSAR A2-13-1	C++14	Rule
AUTOSAR C++14 Rule A2-13-2	String literals with different encoding prefixes shall not be concatenated	AUTOSAR A2-13-2	C++14	Rule
AUTOSAR C++14 Rule A2-13-3	Type wchar_t shall not be used	AUTOSAR A2-13-3	C++14	Rule
AUTOSAR C++14 Rule A2-13-4	String literals shall not be assigned to non-constant pointers	AUTOSAR A2-13-4	C++14	Rule
AUTOSAR C++14 Rule A2-13-5	Hexadecimal constants should be uppercase	AUTOSAR A2-13-5	C++14	Rule
AUTOSAR C++14 Rule A2-13-6	Universal character names shall be used only inside character or string literals	AUTOSAR A2-13-6	C++14	Rule
AUTOSAR C++14 Rule A2-3-1	Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code	AUTOSAR A2-3-1	C++14	Rule
AUTOSAR C++14 Rule A2-5-1	Trigraphs shall not be used	AUTOSAR A2-5-1	C++14	Rule
AUTOSAR C++14 Rule A2-5-2	Digraphs shall not be used	AUTOSAR A2-5-2	C++14	Rule
AUTOSAR C++14 Rule A2-7-1	The character \ shall not occur as a last character of a C++ comment	AUTOSAR A2-7-1	C++14	Rule
AUTOSAR C++14 Rule A2-7-3	All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation	AUTOSAR A2-7-3	C++14	Rule
AUTOSAR C++14 Rule A20-8-1	An already-owned pointer value shall not be stored in an unrelated smart pointer	AUTOSAR A20-8-1	C++14	Rule
AUTOSAR C++14 Rule A20-8-2	A std::unique_ptr shall be used to represent exclusive ownership	AUTOSAR A20-8-2	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A20-8-3	A <code>std::shared_ptr</code> shall be used to represent shared ownership	AUTOSAR A20-8-3	C++14	Rule
AUTOSAR C++14 Rule A20-8-4	A <code>std::unique_ptr</code> shall be used over <code>std::shared_ptr</code> if ownership sharing is not required	AUTOSAR A20-8-4	C++14	Rule
AUTOSAR C++14 Rule A20-8-5	<code>std::make_unique</code> shall be used to construct objects owned by <code>std::unique_ptr</code>	AUTOSAR A20-8-5	C++14	Rule
AUTOSAR C++14 Rule A20-8-6	<code>std::make_shared</code> shall be used to construct objects owned by <code>std::shared_ptr</code>	AUTOSAR A20-8-6	C++14	Rule
AUTOSAR C++14 Rule A21-8-1	Arguments to character-handling functions shall be representable as an unsigned char	AUTOSAR A21-8-1	C++14	Rule
AUTOSAR C++14 Rule A23-0-1	An iterator shall not be implicitly converted to <code>const_iterator</code>	AUTOSAR A23-0-1	C++14	Rule
AUTOSAR C++14 Rule A23-0-2	Elements of a container shall only be accessed via valid references, iterators, and pointers	AUTOSAR A23-0-2	C++14	Rule
AUTOSAR C++14 Rule A25-1-1	Non-static data members or captured values of predicate function objects that are state related to this object's identity shall not be copied	AUTOSAR A25-1-1	C++14	Rule
AUTOSAR C++14 Rule A26-5-1	Pseudorandom numbers shall not be generated using <code>std::rand()</code>	AUTOSAR A26-5-1	C++14	Rule
AUTOSAR C++14 Rule A26-5-2	Random number engines shall not be default-initialized	AUTOSAR A26-5-2	C++14	Rule
AUTOSAR C++14 Rule A27-0-2	A C-style string shall guarantee sufficient space for data and the null terminator	AUTOSAR A27-0-2	C++14	Rule
AUTOSAR C++14 Rule A27-0-3	Alternate input and output operations on a file stream shall not be used without an intervening flush or positioning call	AUTOSAR A27-0-3	C++14	Rule
AUTOSAR C++14 Rule A27-0-4	C-style strings shall not be used	AUTOSAR A27-0-4	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A3-1-1	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule	AUTOSAR A3-1-1	C++14	Rule
AUTOSAR C++14 Rule A3-1-2	Header files, that are defined locally in the project, shall have a file name extension of one of: .h, .hpp or .hxx	AUTOSAR A3-1-2	C++14	Rule
AUTOSAR C++14 Rule A3-1-3	Implementation files, that are defined locally in the project, should have a file name extension of ".cpp"	AUTOSAR A3-1-3	C++14	Rule
AUTOSAR C++14 Rule A3-1-4	When an array with external linkage is declared, its size shall be stated explicitly	AUTOSAR A3-1-4	C++14	Rule
AUTOSAR C++14 Rule A3-1-6	Trivial accessor and mutator functions should be inlined	AUTOSAR A3-1-6	C++14	Rule
AUTOSAR C++14 Rule A3-3-1	Objects or functions with external linkage (including members of named namespaces) shall be declared in a header file	AUTOSAR A3-3-1	C++14	Rule
AUTOSAR C++14 Rule A3-3-2	Static and thread-local objects shall be constant-initialized	AUTOSAR A3-3-2	C++14	Rule
AUTOSAR C++14 Rule A3-9-1	Fixed width integer types from <stdint>, indicating the size and signedness, shall be used in place of the basic numerical types	AUTOSAR A3-9-1	C++14	Rule
AUTOSAR C++14 Rule A4-10-1	Only nullptr literal shall be used as the null-pointer-constraint	AUTOSAR A4-10-1	C++14	Rule
AUTOSAR C++14 Rule A4-5-1	Expressions with type enum or enum class shall not be used as operands to built-in and overloaded operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=	AUTOSAR A4-5-1	C++14	Rule
AUTOSAR C++14 Rule A4-7-1	An integer expression shall not lead to data loss	AUTOSAR A4-7-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits	AUTOSAR A5-0-1	C++14	Rule
AUTOSAR C++14 Rule A5-0-2	The condition of an if-statement and the condition of an iteration statement shall have type bool	AUTOSAR A5-0-2	C++14	Rule
AUTOSAR C++14 Rule A5-0-3	The declaration of objects shall contain no more than two levels of pointer indirection	AUTOSAR A5-0-3	C++14	Rule
AUTOSAR C++14 Rule A5-0-4	Pointer arithmetic shall not be used with pointers to non-final classes	AUTOSAR A5-0-4	C++14	Rule
AUTOSAR C++14 Rule A5-1-2	Variables shall not be implicitly captured in a lambda expression	AUTOSAR A5-1-2	C++14	Rule
AUTOSAR C++14 Rule A5-1-3	Parameter list (possibly empty) shall be included in every lambda expression	AUTOSAR A5-1-3	C++14	Rule
AUTOSAR C++14 Rule A5-1-4	A lambda expression object shall not outlive any of its reference-captured objects	AUTOSAR A5-1-4	C++14	Rule
AUTOSAR C++14 Rule A5-1-6	Return type of a non-void return type lambda expression should be explicitly specified	AUTOSAR A5-1-6	C++14	Rule
AUTOSAR C++14 Rule A5-1-7	A lambda shall not be an operand to decltype or typeid	AUTOSAR A5-1-7	C++14	Rule
AUTOSAR C++14 Rule A5-1-8	Lambda expressions should not be defined inside another lambda expression	AUTOSAR A5-1-8	C++14	Rule
AUTOSAR C++14 Rule A5-1-9	Identical unnamed lambda expressions shall be replaced with a named function or a named lambda expression	AUTOSAR A5-1-9	C++14	Rule
AUTOSAR C++14 Rule A5-10-1	A pointer to member virtual function shall only be tested for equality with null-pointer-constant	AUTOSAR A5-10-1	C++14	Rule
AUTOSAR C++14 Rule A5-16-1	The ternary conditional operator shall not be used as a sub-expression	AUTOSAR A5-16-1	C++14	Rule
AUTOSAR C++14 Rule A5-2-1	dynamic_cast should not be used	AUTOSAR A5-2-1	C++14	Rule
AUTOSAR C++14 Rule A5-2-2	Traditional C-style casts shall not be used	AUTOSAR A5-2-2	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A5-2-3	A cast shall not remove any const or volatile qualification from the type of a pointer or reference	AUTOSAR A5-2-3	C++14	Rule
AUTOSAR C++14 Rule A5-2-4	reinterpret_cast shall not be used	AUTOSAR A5-2-4	C++14	Rule
AUTOSAR C++14 Rule A5-2-5	An array or container shall not be accessed beyond its range	AUTOSAR A5-2-5	C++14	Rule
AUTOSAR C++14 Rule A5-2-6	The operands of a logical && or shall be parenthesized if the operands contain binary operators	AUTOSAR A5-2-6	C++14	Rule
AUTOSAR C++14 Rule A5-3-3	Pointers to incomplete class types shall not be deleted	AUTOSAR A5-3-3	C++14	Rule
AUTOSAR C++14 Rule A5-5-1	A pointer to member shall not access non-existent class members	AUTOSAR A5-5-1	C++14	Rule
AUTOSAR C++14 Rule A5-6-1	The right hand operand of the integer division or remainder operators shall not be equal to zero	AUTOSAR A5-6-1	C++14	Rule
AUTOSAR C++14 Rule A6-2-1	Move and copy assignment operators shall either move or respectively copy base classes and data members of a class, without any side effects	AUTOSAR A6-2-1	C++14	Rule
AUTOSAR C++14 Rule A6-2-2	Expression statements shall not be explicit calls to constructors of temporary objects only	AUTOSAR A6-2-2	C++14	Rule
AUTOSAR C++14 Rule A6-4-1	A switch statement shall have at least two case-clauses, distinct from the default label	AUTOSAR A6-4-1	C++14	Rule
AUTOSAR C++14 Rule A6-5-1	A for-loop that loops through all elements of the container and does not use its loop-counter shall not be used	AUTOSAR A6-5-1	C++14	Rule
AUTOSAR C++14 Rule A6-5-2	A for loop shall contain a single loop-counter which shall not have floating-point type	AUTOSAR A6-5-2	C++14	Rule
AUTOSAR C++14 Rule A6-5-3	Do statements should not be used	AUTOSAR A6-5-3	C++14	Rule
AUTOSAR C++14 Rule A6-5-4	For-init-statement and expression should not perform actions other than loop-counter initialization and modification	AUTOSAR A6-5-4	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A6-6-1	The goto statement shall not be used	AUTOSAR A6-6-1	C++14	Rule
AUTOSAR C++14 Rule A7-1-1	Constexpr or const specifiers shall be used for immutable data declaration	AUTOSAR A7-1-1	C++14	Rule
AUTOSAR C++14 Rule A7-1-2	The constexpr specifier shall be used for values that can be determined at compile time	AUTOSAR A7-1-2	C++14	Rule
AUTOSAR C++14 Rule A7-1-3	CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name	AUTOSAR A7-1-3	C++14	Rule
AUTOSAR C++14 Rule A7-1-4	The register keyword shall not be used	AUTOSAR A7-1-4	C++14	Rule
AUTOSAR C++14 Rule A7-1-5	The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax	AUTOSAR A7-1-5	C++14	Rule
AUTOSAR C++14 Rule A7-1-6	The typedef specifier shall not be used	AUTOSAR A7-1-6	C++14	Rule
AUTOSAR C++14 Rule A7-1-7	Each expression statement and identifier declaration shall be placed on a separate line	AUTOSAR A7-1-7	C++14	Rule
AUTOSAR C++14 Rule A7-1-8	A non-type specifier shall be placed before a type specifier in a declaration	AUTOSAR A7-1-8	C++14	Rule
AUTOSAR C++14 Rule A7-1-9	A class, structure, or enumeration shall not be declared in the definition of its type	AUTOSAR A7-1-9	C++14	Rule
AUTOSAR C++14 Rule A7-2-1	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration	AUTOSAR A7-2-1	C++14	Rule
AUTOSAR C++14 Rule A7-2-2	Enumeration underlying type shall be explicitly defined	AUTOSAR A7-2-2	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A7-2-3	Enumerations shall be declared as scoped enum classes	AUTOSAR A7-2-3	C++14	Rule
AUTOSAR C++14 Rule A7-2-4	In an enumeration, either (1) none, (2) the first or (3) all enumerators shall be initialized	AUTOSAR A7-2-4	C++14	Rule
AUTOSAR C++14 Rule A7-3-1	All overloads of a function shall be visible from where it is called	AUTOSAR A7-3-1	C++14	Rule
AUTOSAR C++14 Rule A7-4-1	The asm declaration shall not be used	AUTOSAR A7-4-1	C++14	Rule
AUTOSAR C++14 Rule A7-5-1	A function shall not return a reference or a pointer to a parameter that is passed by reference to const	AUTOSAR A7-5-1	C++14	Rule
AUTOSAR C++14 Rule A7-5-2	Functions shall not call themselves, either directly or indirectly	AUTOSAR A7-5-2	C++14	Rule
AUTOSAR C++14 Rule A7-6-1	Functions declared with the [[noreturn]] attribute shall not return	AUTOSAR A7-6-1	C++14	Rule
AUTOSAR C++14 Rule A8-2-1	When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters	AUTOSAR A8-2-1	C++14	Rule
AUTOSAR C++14 Rule A8-4-1	Functions shall not be defined using the ellipsis notation	AUTOSAR A8-4-1	C++14	Rule
AUTOSAR C++14 Rule A8-4-10	A parameter shall be passed by reference if it can't be NULL	AUTOSAR A8-4-10	C++14	Rule
AUTOSAR C++14 Rule A8-4-11	A smart pointer shall only be used as a parameter type if it expresses lifetime semantics	AUTOSAR A8-4-11	C++14	Rule
AUTOSAR C++14 Rule A8-4-12	A std::unique_ptr shall be passed to a function as: (1) a copy to express the function assumes ownership (2) an lvalue reference to express that the function replaces the managed object.	AUTOSAR A8-4-12	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A8-4-13	A <code>std::shared_ptr</code> shall be passed to a function as: (1) a copy to express the function shares ownership (2) an lvalue reference to express that the function replaces the managed object (3) a <code>const</code> lvalue reference to express that the function retains a reference count.	AUTOSAR A8-4-13	C++14	Rule
AUTOSAR C++14 Rule A8-4-2	All exit paths from a function with non-void return type shall have an explicit return statement with an expression	AUTOSAR A8-4-2	C++14	Rule
AUTOSAR C++14 Rule A8-4-4	Multiple output values from a function should be returned as a struct or tuple	AUTOSAR A8-4-4	C++14	Rule
AUTOSAR C++14 Rule A8-4-5	"consume" parameters declared as <code>X &&</code> shall always be moved from	AUTOSAR A8-4-5	C++14	Rule
AUTOSAR C++14 Rule A8-4-6	"forward" parameters declared as <code>T &&</code> shall always be forwarded	AUTOSAR A8-4-6	C++14	Rule
AUTOSAR C++14 Rule A8-4-7	"in" parameters for "cheap to copy" types shall be passed by value	AUTOSAR A8-4-7	C++14	Rule
AUTOSAR C++14 Rule A8-4-8	Output parameters shall not be used	AUTOSAR A8-4-8	C++14	Rule
AUTOSAR C++14 Rule A8-4-9	"in-out" parameters declared as <code>T &</code> shall be modified	AUTOSAR A8-4-9	C++14	Rule
AUTOSAR C++14 Rule A8-5-0	All memory shall be initialized before it is read	AUTOSAR A8-5-0	C++14	Rule
AUTOSAR C++14 Rule A8-5-1	In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition	AUTOSAR A8-5-1	C++14	Rule
AUTOSAR C++14 Rule A8-5-2	Braced-initialization <code>{}</code> , without equals sign, shall be used for variable initialization	AUTOSAR A8-5-2	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A8-5-3	A variable of type auto shall not be initialized using {} or ={} braced-initialization	AUTOSAR A8-5-3	C++14	Rule
AUTOSAR C++14 Rule A8-5-4	If a class has a user-declared constructor that takes a parameter of type std::initializer_list, then it shall be the only constructor apart from special member function constructors	AUTOSAR A8-5-4	C++14	Rule
AUTOSAR C++14 Rule A9-5-1	Unions shall not be used	AUTOSAR A9-5-1	C++14	Rule
AUTOSAR C++14 Rule M0-1-1	A project shall not contain unreachable code	AUTOSAR M0-1-1	C++14	Rule
AUTOSAR C++14 Rule M0-1-10	Every defined function should be called at least once	AUTOSAR M0-1-10	C++14	Rule
AUTOSAR C++14 Rule M0-1-2	A project shall not contain infeasible paths	AUTOSAR M0-1-2	C++14	Rule
AUTOSAR C++14 Rule M0-1-3	A project shall not contain unused variables	AUTOSAR M0-1-3	C++14	Rule
AUTOSAR C++14 Rule M0-1-4	A project shall not contain non-volatile POD variables having only one use	AUTOSAR M0-1-4	C++14	Rule
AUTOSAR C++14 Rule M0-1-8	All functions with void return type shall have external side effect(s)	AUTOSAR M0-1-8	C++14	Rule
AUTOSAR C++14 Rule M0-1-9	There shall be no dead code	AUTOSAR M0-1-9	C++14	Rule
AUTOSAR C++14 Rule M0-2-1	An object shall not be assigned to an overlapping object	AUTOSAR M0-2-1	C++14	Rule
AUTOSAR C++14 Rule M10-1-1	Classes should not be derived from virtual bases	AUTOSAR M10-1-1	C++14	Rule
AUTOSAR C++14 Rule M10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy	AUTOSAR M10-1-2	C++14	Rule
AUTOSAR C++14 Rule M10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy	AUTOSAR M10-1-3	C++14	Rule
AUTOSAR C++14 Rule M10-2-1	All accessible entity names within a multiple inheritance hierarchy should be unique	AUTOSAR M10-2-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual	AUTOSAR M10-3-3	C++14	Rule
AUTOSAR C++14 Rule M11-0-1	Member data in non-POD class types shall be private	AUTOSAR M11-0-1	C++14	Rule
AUTOSAR C++14 Rule M12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor	AUTOSAR M12-1-1	C++14	Rule
AUTOSAR C++14 Rule M14-5-3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter	AUTOSAR M14-5-3	C++14	Rule
AUTOSAR C++14 Rule M14-6-1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->	AUTOSAR M14-6-1	C++14	Rule
AUTOSAR C++14 Rule M15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement	AUTOSAR M15-0-3	C++14	Rule
AUTOSAR C++14 Rule M15-1-1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown	AUTOSAR M15-1-1	C++14	Rule
AUTOSAR C++14 Rule M15-1-2	NULL shall not be thrown explicitly	AUTOSAR M15-1-2	C++14	Rule
AUTOSAR C++14 Rule M15-1-3	An empty throw (throw;) shall only be used in the compound statement of a catch handler	AUTOSAR M15-1-3	C++14	Rule
AUTOSAR C++14 Rule M15-3-1	Exceptions shall be raised only after start-up and before termination	AUTOSAR M15-3-1	C++14	Rule
AUTOSAR C++14 Rule M15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases	AUTOSAR M15-3-3	C++14	Rule
AUTOSAR C++14 Rule M15-3-4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point	AUTOSAR M15-3-4	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class	AUTOSAR M15-3-6	C++14	Rule
AUTOSAR C++14 Rule M15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last	AUTOSAR M15-3-7	C++14	Rule
AUTOSAR C++14 Rule M16-0-1	#include directives in a file shall only be preceded by other preprocessor directives or comments	AUTOSAR M16-0-1	C++14	Rule
AUTOSAR C++14 Rule M16-0-2	Macros shall only be #define'd or #undef'd in the global namespace	AUTOSAR M16-0-2	C++14	Rule
AUTOSAR C++14 Rule M16-0-5	Arguments to a function-like macro shall not contain tokens that look like pre-processing directives	AUTOSAR M16-0-5	C++14	Rule
AUTOSAR C++14 Rule M16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##	AUTOSAR M16-0-6	C++14	Rule
AUTOSAR C++14 Rule M16-0-7	Undefined macro identifiers shall not be used in #if or #elif pre-processor directives, except as operands to the defined operator	AUTOSAR M16-0-7	C++14	Rule
AUTOSAR C++14 Rule M16-0-8	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token	AUTOSAR M16-0-8	C++14	Rule
AUTOSAR C++14 Rule M16-1-1	The defined pre-processor operator shall only be used in one of the two standard forms	AUTOSAR M16-1-1	C++14	Rule
AUTOSAR C++14 Rule M16-1-2	All #else, #elif and #endif pre-processor directives shall reside in the same file as the #if or #ifdef directive to which they are related	AUTOSAR M16-1-2	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M16-2-3	Include guards shall be provided	AUTOSAR M16-2-3	C++14	Rule
AUTOSAR C++14 Rule M16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition	AUTOSAR M16-3-1	C++14	Rule
AUTOSAR C++14 Rule M16-3-2	The # and ## operators should not be used	AUTOSAR M16-3-2	C++14	Rule
AUTOSAR C++14 Rule M17-0-2	The names of standard library macros and objects shall not be reused	AUTOSAR M17-0-2	C++14	Rule
AUTOSAR C++14 Rule M17-0-3	The names of standard library functions shall not be overridden	AUTOSAR M17-0-3	C++14	Rule
AUTOSAR C++14 Rule M17-0-5	The setjmp macro and the longjmp function shall not be used	AUTOSAR M17-0-5	C++14	Rule
AUTOSAR C++14 Rule M18-0-3	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used	AUTOSAR M18-0-3	C++14	Rule
AUTOSAR C++14 Rule M18-0-4	The time handling functions of library <ctime> shall not be used	AUTOSAR M18-0-4	C++14	Rule
AUTOSAR C++14 Rule M18-0-5	The unbounded functions of library <cstring> shall not be used	AUTOSAR M18-0-5	C++14	Rule
AUTOSAR C++14 Rule M18-2-1	The macro offsetof shall not be used	AUTOSAR M18-2-1	C++14	Rule
AUTOSAR C++14 Rule M18-7-1	The signal handling facilities of <csignal> shall not be used	AUTOSAR M18-7-1	C++14	Rule
AUTOSAR C++14 Rule M19-3-1	The error indicator errno shall not be used	AUTOSAR M19-3-1	C++14	Rule
AUTOSAR C++14 Rule M2-10-1	Different identifiers shall be typographically unambiguous	AUTOSAR M2-10-1	C++14	Rule
AUTOSAR C++14 Rule M2-13-2	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used	AUTOSAR M2-13-2	C++14	Rule
AUTOSAR C++14 Rule M2-13-3	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type	AUTOSAR M2-13-3	C++14	Rule
AUTOSAR C++14 Rule M2-13-4	Literal suffixes shall be upper case	AUTOSAR M2-13-4	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M2-7-1	The character sequence /* shall not be used within a C-style comment	AUTOSAR M2-7-1	C++14	Rule
AUTOSAR C++14 Rule M27-0-1	The stream input/output library <cstdio> shall not be used	AUTOSAR M27-0-1	C++14	Rule
AUTOSAR C++14 Rule M3-1-2	Functions shall not be declared at block scope	AUTOSAR M3-1-2	C++14	Rule
AUTOSAR C++14 Rule M3-2-1	All declarations of an object or function shall have compatible types	AUTOSAR M3-2-1	C++14	Rule
AUTOSAR C++14 Rule M3-2-2	The One Definition Rule shall not be violated	AUTOSAR M3-2-2	C++14	Rule
AUTOSAR C++14 Rule M3-2-3	A type, object or function that is used in multiple translation units shall be declared in one and only one file	AUTOSAR M3-2-3	C++14	Rule
AUTOSAR C++14 Rule M3-2-4	An identifier with external linkage shall have exactly one definition	AUTOSAR M3-2-4	C++14	Rule
AUTOSAR C++14 Rule M3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier	AUTOSAR M3-3-2	C++14	Rule
AUTOSAR C++14 Rule M3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility	AUTOSAR M3-4-1	C++14	Rule
AUTOSAR C++14 Rule M3-9-1	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations	AUTOSAR M3-9-1	C++14	Rule
AUTOSAR C++14 Rule M3-9-3	The underlying bit representations of floating-point values shall not be used	AUTOSAR M3-9-3	C++14	Rule
AUTOSAR C++14 Rule M4-10-1	NULL shall not be used as an integer value	AUTOSAR M4-10-1	C++14	Rule
AUTOSAR C++14 Rule M4-10-2	Literal zero (0) shall not be used as the null-pointer-constant	AUTOSAR M4-10-2	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M4-5-1	Expressions with type <code>bool</code> shall not be used as operands to built-in operators other than the assignment operator <code>=</code> , the logical operators <code>&&</code> , <code> </code> , <code>!</code> , the equality operators <code>==</code> and <code>!=</code> , the unary <code>&</code> operator, and the conditional operator	AUTOSAR M4-5-1	C++14	Rule
AUTOSAR C++14 Rule M4-5-3	Expressions with type <code>(plain) char</code> and <code>wchar_t</code> shall not be used as operands to built-in operators other than the assignment operator <code>=</code> , the equality operators <code>==</code> and <code>!=</code> , and the unary <code>&</code> operator	AUTOSAR M4-5-3	C++14	Rule
AUTOSAR C++14 Rule M5-0-10	If the bitwise operators <code>~</code> and <code><<</code> are applied to an operand with an underlying type of <code>unsigned char</code> or <code>unsigned short</code> , the result shall be immediately cast to the underlying type of the operand	AUTOSAR M5-0-10	C++14	Rule
AUTOSAR C++14 Rule M5-0-11	The plain <code>char</code> type shall only be used for the storage and use of character values	AUTOSAR M5-0-11	C++14	Rule
AUTOSAR C++14 Rule M5-0-12	Signed <code>char</code> and unsigned <code>char</code> type shall only be used for the storage and use of numeric values	AUTOSAR M5-0-12	C++14	Rule
AUTOSAR C++14 Rule M5-0-14	The first operand of a conditional-operator shall have type <code>bool</code>	AUTOSAR M5-0-14	C++14	Rule
AUTOSAR C++14 Rule M5-0-15	Array indexing shall be the only form of pointer arithmetic	AUTOSAR M5-0-15	C++14	Rule
AUTOSAR C++14 Rule M5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array	AUTOSAR M5-0-16	C++14	Rule
AUTOSAR C++14 Rule M5-0-17	Subtraction between pointers shall only be applied to pointers that address elements of the same array	AUTOSAR M5-0-17	C++14	Rule
AUTOSAR C++14 Rule M5-0-18	<code>></code> , <code>>=</code> , <code><</code> , <code><=</code> shall not be applied to objects of pointer type, except where they point to the same array	AUTOSAR M5-0-18	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M5-0-20	Non-constant operands to a binary bitwise operator shall have the same underlying type	AUTOSAR M5-0-20	C++14	Rule
AUTOSAR C++14 Rule M5-0-21	Bitwise operators shall only be applied to operands of unsigned underlying type	AUTOSAR M5-0-21	C++14	Rule
AUTOSAR C++14 Rule M5-0-3	A cvalue expression shall not be implicitly converted to a different underlying type	AUTOSAR M5-0-3	C++14	Rule
AUTOSAR C++14 Rule M5-0-4	An implicit integral conversion shall not change the signedness of the underlying type	AUTOSAR M5-0-4	C++14	Rule
AUTOSAR C++14 Rule M5-0-5	There shall be no implicit floating-integral conversions	AUTOSAR M5-0-5	C++14	Rule
AUTOSAR C++14 Rule M5-0-6	An implicit integral or floating-point conversion shall not reduce the size of the underlying type	AUTOSAR M5-0-6	C++14	Rule
AUTOSAR C++14 Rule M5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression	AUTOSAR M5-0-7	C++14	Rule
AUTOSAR C++14 Rule M5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression	AUTOSAR M5-0-8	C++14	Rule
AUTOSAR C++14 Rule M5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression	AUTOSAR M5-0-9	C++14	Rule
AUTOSAR C++14 Rule M5-14-1	The right hand operand of a logical &&, operators shall not contain side effects	AUTOSAR M5-14-1	C++14	Rule
AUTOSAR C++14 Rule M5-18-1	The comma operator shall not be used	AUTOSAR M5-18-1	C++14	Rule
AUTOSAR C++14 Rule M5-19-1	Evaluation of constant unsigned integer expressions shall not lead to wrap-around	AUTOSAR M5-19-1	C++14	Rule
AUTOSAR C++14 Rule M5-2-10	The increment (++) and decrement (--) operators shall not be mixed with other operators in an expression	AUTOSAR M5-2-10	C++14	Rule
AUTOSAR C++14 Rule M5-2-11	The comma operator, && operator and the operator shall not be overloaded	AUTOSAR M5-2-11	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M5-2-12	An identifier with array type passed as a function argument shall not decay to a pointer	AUTOSAR M5-2-12	C++14	Rule
AUTOSAR C++14 Rule M5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of <code>dynamic_cast</code>	AUTOSAR M5-2-2	C++14	Rule
AUTOSAR C++14 Rule M5-2-3	Casts from a base class to a derived class should not be performed on polymorphic types	AUTOSAR M5-2-3	C++14	Rule
AUTOSAR C++14 Rule M5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type	AUTOSAR M5-2-6	C++14	Rule
AUTOSAR C++14 Rule M5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type	AUTOSAR M5-2-8	C++14	Rule
AUTOSAR C++14 Rule M5-2-9	A cast shall not convert a pointer type to an integral type	AUTOSAR M5-2-9	C++14	Rule
AUTOSAR C++14 Rule M5-3-1	Each operand of the <code>!</code> operator, the logical <code>&&</code> or the logical <code> </code> operators shall have type <code>bool</code>	AUTOSAR M5-3-1	C++14	Rule
AUTOSAR C++14 Rule M5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned	AUTOSAR M5-3-2	C++14	Rule
AUTOSAR C++14 Rule M5-3-3	The unary <code>&</code> operator shall not be overloaded	AUTOSAR M5-3-3	C++14	Rule
AUTOSAR C++14 Rule M5-3-4	Evaluation of the operand to the <code>sizeof</code> operator shall not contain side effects	AUTOSAR M5-3-4	C++14	Rule
AUTOSAR C++14 Rule M6-2-1	Assignment operators shall not be used in sub-expressions	AUTOSAR M6-2-1	C++14	Rule
AUTOSAR C++14 Rule M6-2-3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character	AUTOSAR M6-2-3	C++14	Rule
AUTOSAR C++14 Rule M6-3-1	The statement forming the body of a <code>switch</code> , <code>while</code> , <code>do ... while</code> or <code>for</code> statement shall be a compound statement	AUTOSAR M6-3-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M6-4-1	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement	AUTOSAR M6-4-1	C++14	Rule
AUTOSAR C++14 Rule M6-4-2	All if ... else if constructs shall be terminated with an else clause	AUTOSAR M6-4-2	C++14	Rule
AUTOSAR C++14 Rule M6-4-3	A switch statement shall be a well-formed switch statement	AUTOSAR M6-4-3	C++14	Rule
AUTOSAR C++14 Rule M6-4-4	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	AUTOSAR M6-4-4	C++14	Rule
AUTOSAR C++14 Rule M6-4-5	An unconditional throw or break statement shall terminate every non-empty switch-clause	AUTOSAR M6-4-5	C++14	Rule
AUTOSAR C++14 Rule M6-4-6	The final clause of a switch statement shall be the default-clause	AUTOSAR M6-4-6	C++14	Rule
AUTOSAR C++14 Rule M6-4-7	The condition of a switch statement shall not have bool type	AUTOSAR M6-4-7	C++14	Rule
AUTOSAR C++14 Rule M6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=	AUTOSAR M6-5-2	C++14	Rule
AUTOSAR C++14 Rule M6-5-3	The loop-counter shall not be modified within condition or statement	AUTOSAR M6-5-3	C++14	Rule
AUTOSAR C++14 Rule M6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop	AUTOSAR M6-5-4	C++14	Rule
AUTOSAR C++14 Rule M6-5-5	A loop-control-variable other than the loop-counter shall not be modified within condition or expression	AUTOSAR M6-5-5	C++14	Rule
AUTOSAR C++14 Rule M6-5-6	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool	AUTOSAR M6-5-6	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement	AUTOSAR M6-6-1	C++14	Rule
AUTOSAR C++14 Rule M6-6-2	The goto statement shall jump to a label declared later in the same function body	AUTOSAR M6-6-2	C++14	Rule
AUTOSAR C++14 Rule M6-6-3	The continue statement shall only be used within a well-formed for loop	AUTOSAR M6-6-3	C++14	Rule
AUTOSAR C++14 Rule M7-1-2	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified	AUTOSAR M7-1-2	C++14	Rule
AUTOSAR C++14 Rule M7-3-1	The global namespace shall only contain main, namespace declarations and extern "C" declarations	AUTOSAR M7-3-1	C++14	Rule
AUTOSAR C++14 Rule M7-3-2	The identifier main shall not be used for a function other than the global function main	AUTOSAR M7-3-2	C++14	Rule
AUTOSAR C++14 Rule M7-3-3	There shall be no unnamed namespaces in header files	AUTOSAR M7-3-3	C++14	Rule
AUTOSAR C++14 Rule M7-3-4	Using-directives shall not be used	AUTOSAR M7-3-4	C++14	Rule
AUTOSAR C++14 Rule M7-3-6	Using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files	AUTOSAR M7-3-6	C++14	Rule
AUTOSAR C++14 Rule M7-4-2	Assembler instructions shall only be introduced using the asm declaration	AUTOSAR M7-4-2	C++14	Rule
AUTOSAR C++14 Rule M7-4-3	Assembly language shall be encapsulated and isolated	AUTOSAR M7-4-3	C++14	Rule
AUTOSAR C++14 Rule M8-0-1	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively	AUTOSAR M8-0-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule M8-3-1	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments	AUTOSAR M8-3-1	C++14	Rule
AUTOSAR C++14 Rule M8-4-2	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration	AUTOSAR M8-4-2	C++14	Rule
AUTOSAR C++14 Rule M8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &	AUTOSAR M8-4-4	C++14	Rule
AUTOSAR C++14 Rule M8-5-2	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures	AUTOSAR M8-5-2	C++14	Rule
AUTOSAR C++14 Rule M9-3-1	Const member functions shall not return non-const pointers or references to class-data	AUTOSAR M9-3-1	C++14	Rule
AUTOSAR C++14 Rule M9-3-3	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const	AUTOSAR M9-3-3	C++14	Rule
AUTOSAR C++14 Rule M9-6-4	Named bit-fields with signed integer type shall have a length of more than one bit	AUTOSAR M9-6-4	C++14	Rule

Partially Automated Rules

According to the AUTOSAR C++14 standard, static analysis detects only a subset of all possible violation of **Partially Automated** rules. Polyspace Bug Finder supports 22 out of 22 **Partially Automated** rules. For details about which error scenarios of a rule Polyspace detects, see the **Polyspace Implementation** section in the reference page of the rule.

Polyspace supports these **Partially Automated** rules.

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A0-4-4	Range, domain and pole errors shall be checked when using math functions	AUTOSAR A0-4-4	C++14	Rule
AUTOSAR C++14 Rule A12-0-2	Bitwise operations and operations that assume data representation in memory shall not be performed on objects	AUTOSAR A12-0-2	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A12-1-5	Common class initialization for non-constant members shall be done by a delegating constructor	AUTOSAR A12-1-5	C++14	Rule
AUTOSAR C++14 Rule A12-8-3	Moved-from object shall not be read-accessed	AUTOSAR A12-8-3	C++14	Rule
AUTOSAR C++14 Rule A14-5-2	Class members that are not dependent on template class parameters should be defined in a separate base class	AUTOSAR A14-5-2	C++14	Rule
AUTOSAR C++14 Rule A15-0-2	At least the basic guarantee for exception safety shall be provided for all operations. In addition, each function may offer either the strong guarantee or the nothrow guarantee	AUTOSAR A15-0-2	C++14	Rule
AUTOSAR C++14 Rule A15-0-7	Exception handling mechanism shall guarantee a deterministic worst-case time execution time	AUTOSAR A15-0-7	C++14	Rule
AUTOSAR C++14 Rule A15-1-4	If a function exits with an exception, then before a throw, the function shall place all objects/resources that the function constructed in valid states or it shall delete them.	AUTOSAR A15-1-4	C++14	Rule
AUTOSAR C++14 Rule A15-2-2	If a constructor is not noexcept and the constructor cannot finish object initialization, then it shall deallocate the object's resources and it shall throw an exception	AUTOSAR A15-2-2	C++14	Rule
AUTOSAR C++14 Rule A15-3-3	Main function and a task main function shall catch at least: base class exceptions from all third-party libraries used, std::exception and all otherwise unhandled exceptions	AUTOSAR A15-3-3	C++14	Rule
AUTOSAR C++14 Rule A15-5-2	Program shall not be abruptly terminated. In particular, an implicit or explicit invocation of std::abort(), std::quick_exit(), std::_Exit(), std::terminate() shall not be done	AUTOSAR A15-5-2	C++14	Rule
AUTOSAR C++14 Rule A18-5-2	Non-placement new or delete expressions shall not be used	AUTOSAR A18-5-2	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker		
AUTOSAR C++14 Rule A18-5-5	Memory management functions shall ensure the following: (a) deterministic behavior resulting with the existence of worst-case execution time, (b) avoiding memory fragmentation, (c) avoid running out of memory, (d) avoiding mismatched allocations or deallocations, (e) no dependence on non-deterministic calls to kernel	AUTOSAR A18-5-5	C++14	Rule
AUTOSAR C++14 Rule A18-5-8	Objects that do not outlive a function shall have automatic storage duration	AUTOSAR A18-5-8	C++14	Rule
AUTOSAR C++14 Rule A3-1-5	A function definition shall only be placed in a class definition if (1) the function is intended to be inlined (2) it is a member function template (3) it is a member function of a class template	AUTOSAR A3-1-5	C++14	Rule
AUTOSAR C++14 Rule A5-1-1	Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead	AUTOSAR A5-1-1	C++14	Rule
AUTOSAR C++14 Rule A5-3-2	Null pointers shall not be dereferenced	AUTOSAR A5-3-2	C++14	Rule
AUTOSAR C++14 Rule A9-3-1	Member functions shall not return non-constant "raw" pointers or references to private or protected data owned by the class	AUTOSAR A9-3-1	C++14	Rule
AUTOSAR C++14 Rule A9-6-1	Data types used for interfacing with hardware or conforming to communication protocols shall be trivial, standard-layout and only contain members of types with defined sizes	AUTOSAR A9-6-1	C++14	Rule
AUTOSAR C++14 Rule M5-0-2	Limited dependence should be placed on C++ operator precedence rules in expressions	AUTOSAR M5-0-2	C++14	Rule
AUTOSAR C++14 Rule M5-8-1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand	AUTOSAR M5-8-1	C++14	Rule

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality	AUTOSAR C++14 Rule M6-2-2

See Also

Check AUTOSAR C++ 14 (-autosar-cpp14)

More About

- “Check for and Review Coding Standard Violations” on page 16-2
- “Coding Standards”
- “Required AUTOSAR C++14 Coding Rules Supported by Polyspace Bug Finder” on page 17-125
- “Required MISRA C++:2008 Coding Rules Supported by Polyspace Bug Finder” on page 17-83
- “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78

Configure Bug Finder Checkers

- “Choose Specific Bug Finder Defect Checkers” on page 18-2
- “Modify Default Behavior of Bug Finder Checkers” on page 18-3
- “Modify Bug Finder Checkers Through Code Behavior Specifications” on page 18-12
- “Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers” on page 18-21
- “Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries” on page 18-24
- “Extend Bug Finder Checkers to Find Defects from Specific System Input Values” on page 18-26
- “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments” on page 18-30
- “Extend Checkers for Initialization to Check Function Arguments Passed by Pointers” on page 18-33
- “Extend Data Race Checkers to Atomic Operations” on page 18-35
- “Prepare Checkers Configuration for Polyspace Bug Finder Analysis” on page 18-38
- “Bug Finder Defect Groups” on page 18-43
- “Classification of Defects by Impact” on page 18-49
- “Sources of Tainting in a Polyspace Analysis” on page 18-61
- “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 18-65
- “Polyspace Bug Finder Defects Checkers Enabled by Default for Generated Code” on page 18-70
- “Bug Finder Results Found in Fast Analysis Mode” on page 18-72
- “Extend CWE Coding Standard Coverage Using Polyspace Defect Checkers” on page 18-97

Choose Specific Bug Finder Defect Checkers

You can check your C/C++ code using the predefined subsets of defect checkers in Bug Finder. However, you can also customize which defects to check for during the analysis.

You can use a spreadsheet to keep track of the defect checkers that you enable and add notes explaining why you do not enable the other checkers. A spreadsheet of checkers is provided in *polyspaceroot*\polyspace\resources. Here, *polyspaceroot* is the Polyspace installation folder, such as C:\Program Files\Polyspace\R2019a.

User Interface (Desktop Products Only)

- 1 On the **Configuration** pane, select **Bug Finder Analysis**.
- 2 From the **Find defects** menu, select a set of defects. The options are:
 - **default** for the default list of defects. This list contains defects that are applicable to most coding projects.

See “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 18-65.
 - **all** for all defects.
 - **CWE** for defects related to CWE coding standard.

For more information, see “Extend CWE Coding Standard Coverage Using Polyspace Defect Checkers” on page 18-97.
 - **custom** to add defects to the default list or remove defects from it.

To standardize the bug finding across your organization, you can save your list of defect checkers as a configuration template and share with others. See “Create Project in Polyspace Desktop User Interface Using Configuration Template” on page 2-13.

Command Line

Use the option `Find defects (-checkers)`. Specify a comma-separated list of checkers as arguments. For instance, to run a Bug Finder analysis on a server with only the data race checkers enabled, enter:

```
polyspace-bug-finder-server -sources filename -checkers DATA_RACE,DATA_RACE_STD_LIB
```

Use short names for the Bug Finder checkers instead of their full names. See “Short Names of Bug Finder Defect Groups and Defect Checkers” on page 30-11.

See Also

`Find defects (-checkers)`

More About

- “Bug Finder Defect Groups” on page 18-43
- “Short Names of Bug Finder Defect Groups and Defect Checkers” on page 30-11

Modify Default Behavior of Bug Finder Checkers

A Polyspace Bug Finder analysis checks C/C++ code for bugs and external coding standard violations. By default, the Bug Finder checkers are designed to:

- Show as few false positives as possible.
- Require minimal setup upfront.

However, for specific projects, you might want to modify the default behavior of some checkers. For instance, you might want to treat some user defined data types as effectively boolean or detect data races involving operations that Bug Finder considers as atomic by default.

Use this topic to find the modifications allowed for Bug Finder checkers. Alternatively, you can search for these options in the analysis report to see if the default behavior of checkers were modified.

Note that:

- The options do not enable or disable a checker.

To enable or disable specific checkers, see “Choose Specific Bug Finder Defect Checkers” on page 18-2.

- You can use these options solely to modify the behavior of an existing checker.

Options such as target processor type, multitasking options and external constraints can also modify the behavior of a checker. However, the modification happens as a side effect. You typically specify these options to accurately reflect your target environment.

Defect Checkers and Coding Rules Modified by Analysis Options

When using Bug Finder, use the analysis option in two different ways. You can:

- Modify how a checker behaves.
- Modify how Polyspace interprets the code behavior.

Modify Checker Behavior

Use the following options to change the default behavior of Bug Finder defects and coding rule checkers. For instance, use `-checks-using-system-input-values` to run a stricter check.

Option	Option Value	Checkers Modified	Modification
Run stricter checks considering all values of system inputs (<code>-checks-using-system-input-values</code>)		Checkers and coding rules that rely on numerical values of system inputs. See “Checkers That Can Be Extended” on page 18-27.	See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values” on page 18-26.

Option	Option Value	Checkers Modified	Modification
-consider-analysis-perimeter-as-trust-boundary		Tainted Data Defects and related coding rule violations. See “Polyspace Tainted Data Checkers” on page 18-63.	<p>By default, the tainted data defects consider externally obtained data as tainted. By using this option, the following are also considered as tainted data:</p> <ul style="list-style-type: none"> • Formal parameters of externally visible function that do not have a visible caller. • Return values of stubbed functions. • Global variables external to the unit. <p>See “Sources of Tainting in a Polyspace Analysis” on page 18-61</p>
Check guidelines (-guidelines)	<p>XML file.</p> <p>Entries in the XML file define the thresholds for software complexity checkers</p>	Software Complexity Guidelines	See “Reduce Software Complexity by Using Polyspace Checkers” on page 16-11
-detect-atomic-data-race		<ul style="list-style-type: none"> • CERT C: Rule CON43-C • CERT C++: CON43-C • Data race 	<p>By default, these checkers flags data races involving non-atomic operations. If an operation is atomic, it cannot be interrupted by operations in another task or thread. If you use this option, all operations are considered when flagging data races.</p> <p>See also “Extend Data Race Checkers to Atomic Operations” on page 18-35.</p>

Option	Option Value	Checkers Modified	Modification
-detect-bad-float-op-on-zero		Checkers and coding rules related to floating point equality checks, such as Floating point comparison with equality operators	By default, the checker ignores floating point comparisons with equality operators if one of the operands is 0.0. If you use this option, comparisons with 0.0 are also flagged.

Modify Polyspace Interpretation of Code

Use these options to modify how Polyspace interprets the code. For instance, use `-boolean-types` to indicate that certain data types in your code are effectively Boolean or use the option `-code-behavior-specification` to specify certain behaviors of your code.

Option	Option Value	Checkers Modified	Modification
Effective boolean types (<code>-boolean-types</code>)	Data types	<ul style="list-style-type: none"> MISRA C:2004 rules 12.6, 13.2, 15.4 MISRA C:2012 rules 10.1, 10.3, 10.5, 14.4, 16.7 	The rules covered by these checkers involve boolean types. If you use this option, you can treat user-defined types as effectively boolean.
Consider non finite floats (<code>-allow-non-finite-floats</code>)		Checkers and coding rules related to float numerical operations. See <code>Consider non finite floats (-allow-non-finite-floats)</code> .	If you use this option, Polyspace incorporates infinities and NaNs in floating point checks. Checkers that flag infinities and NaNs in floating point operations might be disabled or flag less defects.
Allowed pragmas (<code>-allowed-pragmas</code>)	Names of pragmas	MISRA C:2004 rule 3.4 and MISRA C++ rule 16-6-1	These rules require that all pragma directives must be documented within the compiler documentation. If you use this option, the analysis considers the pragmas specified as documented pragmas.
<code>-code-behavior-specifications</code>	XML file. Entries in the XML file map user-defined functions to functions from the Standard Library.	Checkers and coding rules that detect issues with Standard Library functions. See “Checkers That Can Be Extended” on page 18-25.	See “Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries” on page 18-24.

Option	Option Value	Checkers Modified	Modification
	<p>XML file.</p> <p>Entries in the XML file map user-defined concurrency primitives to ones that Bug Finder can automatically detect.</p>	<p>Concurrency defects and related coding rules. See “Checkers That Can Be Extended” on page 18-31.</p>	<p>See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments” on page 18-30.</p>
	<p>XML file.</p> <p>Entries in the XML file list functions, keywords or macros that you want to prohibit from your source code.</p>	<p>See “Checkers That Can Be Extended” on page 18-22.</p>	<p>See “Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers” on page 18-21.</p>
	<p>XML file.</p> <p>Entries in the XML file list functions whose pointer arguments must point to initialized buffers.</p>	<p>See “Checkers That Can Be Extended” on page 18-34</p>	<p>See “Extend Checkers for Initialization to Check Function Arguments Passed by Pointers” on page 18-33.</p>

Option	Option Value	Checkers Modified	Modification
	<p>XML file.</p> <p>Entries in the XML file define limits on global aspects of your program such as maximum depth of nesting in control flow statements.</p>	<p>MISRA C: 2012 Rule 1.1</p>	<p>You can increase or decrease these parameters of the rule checker:</p> <ul style="list-style-type: none"> • Maximum depth of nesting allowed in control flow statements • Maximum levels of inclusion allowed using include files • Maximum number of constants allowed in an enumeration • Maximum number of macros allowed in a translation unit • Maximum number of members allowed in a structure • Maximum levels of nesting allowed in a structure <p>See “Modify Bug Finder Checkers Through Code Behavior Specifications” on page 18-12.</p>

Option	Option Value	Checkers Modified	Modification
	<p>XML file.</p> <p>Entries in the XML file define how many characters are compared before considering two identifiers as distinct.</p>	<p>MISRA C: 2012 Rules 5.1 to 5.5</p>	<p>These rules require uniqueness of certain types of identifiers. For instance, rule 5.1 requires that external identifiers be distinct.</p> <p>If the difference between two identifiers occurs beyond the first <i>num</i> characters, the rule checker considers the identifiers as identical. You can modify the parameter <i>num</i> separately for external and internal identifiers.</p> <p>See “Modify Bug Finder Checkers Through Code Behavior Specifications” on page 18-12.</p>

Option	Option Value	Checkers Modified	Modification
	<p>XML file.</p> <p>Entries in the XML file defines functions as real-time functions and functions that manages dynamic memory.</p>	<p>AUTOSAR C++14 Rule A18-5-7</p>	<p>This rule requires that dynamic memory allocation and deallocation does not occur in real time. Polyspace assumes that no function in your code runs at real-time, and only certain standard library functions manages dynamic memory.</p> <p>Specify a function in your code as a real-time function by using the behavior <code>REAL_TIME_FUNC</code>.</p> <p>Specify a function in your code as a function that allocates or deallocates dynamic memory by using the behavior <code>MANAGES_MEMORY</code>.</p> <p>See “Modify Bug Finder Checkers Through Code Behavior Specifications” on page 18-12.</p>

Option	Option Value	Checkers Modified	Modification
	<p>XML file.</p> <p>Entries in the XML file defines functions that handle exceptions or manage dynamic memory.</p>	<p>AUTOSAR C++14 Rule A15-0-7</p>	<p>This rule requires that exception handling functions must have a deterministic worst-case execution time. Polyspace recognizes the commonly used exception handling functions and dynamic memory management functions.</p> <p>Specify a function in your code as a function that allocates or deallocates dynamic memory by using the behavior <code>MANAGES_MEMORY</code>.</p> <p>Specify a function in your code as a function that handles exceptions by using the behavior <code>EXCEPTION_HANDLING</code>.</p> <p>See “Modify Bug Finder Checkers Through Code Behavior Specifications” on page 18-12.</p>
	<p>XML file.</p> <p>Entries in the XML file defines critical data members.</p>	<p>Critical data member is not private</p>	<p>This checker flags the critical data members that are declared <code>public</code>. By default, Polyspace assumes that no data member is critical.</p> <p>Specify the critical data members in your code by using the behavior <code>CRITICAL_DATA</code>.</p> <p>See “Modify Bug Finder Checkers Through Code Behavior Specifications” on page 18-12.</p>

Option	Option Value	Checkers Modified	Modification
	Datalog file. Entries in the datalog file define functions relevant to SQL or LDAP injection checkers.	SQL injection and LDAP injection	Specify custom functions that are sensitive to SQL or LDAP injection, functions that sanitize SQL or LDAP queries, and so on. See “Modify Bug Finder Checkers Through Code Behavior Specifications” on page 18-12.

See Also

More About

- “Choose Specific Bug Finder Defect Checkers” on page 18-2
- “Modify Bug Finder Checkers Through Code Behavior Specifications” on page 18-12
- “Bug Finder Defect Groups” on page 18-43

Modify Bug Finder Checkers Through Code Behavior Specifications

Polyspace Bug Finder checks C/C++ code for defects or coding rule violations. To find these issues, some Bug Finder checkers require additional information outside your code.

You can specify most external information using the option `-code-behavior-specifications`. The option allows you to associate specific behaviors with elements in your code. For instance, you can map a custom library function to a standard function or specify that a function acts as a memory management function.

The option takes a file in one of these formats as argument:

- XML
- Datalog

The XML and Datalog format support different code behaviors. This topic describes the checker modifications supported with each file format. For the full list of options that can modify checkers, see “Modify Default Behavior of Bug Finder Checkers” on page 18-3.

XML Format

The code behavior specifications XML file associates specific behaviors with functions (or defines behaviors at global scope). The following sections list behaviors that are relevant to a Bug Finder analysis. For behaviors relevant to Code Prover, see “Modify Code Prover Run-Time Checks Through Code Behavior Specifications” (Polyspace Code Prover).

You can also see the supported behaviors in the sample template file `code-behavior-specifications-template.xml` provided with a Polyspace installation. The file is in `polyspaceroot\polyspace\verifier\cxx\` where `polyspaceroot` is the Polyspace installation folder.

Mapping to Standard Function for Precision Improvement

XML Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
  <functions>
    <function name="custom_function" std="std_function">
      </function>
    </functions>
  </specifications>
```

Use this entry in the XML file to reduce the number of false negatives in a Bug Finder analysis. Sometimes, the analysis does not model certain kinds of functions precisely because of inherent limitations in static verification. In those cases, if you find a standard function that is a close analog of your function, use this mapping. Though your function itself is not analyzed, the analysis is more precise at the locations where you call the function. For instance, if the analysis cannot model your function `cos32` precisely and considers full range for its return value, map it to the `cos` function for a return value in `[-1,1]`.

The analysis ignores the body of your function but emulates your function behavior in the following ways:

- The analysis assumes the same return values for your function as the standard function.

For instance, if you map your function `cos32` to the standard function `cos`, the verification assumes that `cos32` returns values in `[-1,1]`.

- The analysis checks for the same issues as it checks with the standard function.

For instance, if you map your function `acos32` to the standard function `acos`, the `Invalid use of standard library routine` check determines if the argument of `acos32` is in `[-1,1]`.

The functions that you can map to include:

- Standard library functions from `math.h`.
- Memory management functions from `string.h`.

If you run Code Prover, you can map to other functions to improve the verification precision. See “Modify Code Prover Run-Time Checks Through Code Behavior Specifications” (Polyspace Code Prover).

See also “Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries” on page 18-24.

Mapping to Standard Function for Concurrency Detection

XML Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
  <functions>
    <function name="custom_function" std="std_function">
      </function>
    </functions>
  </specifications>
```

Use this entry in the XML file for automatic detection of thread-creation functions and functions that begin and end critical sections. Polyspace supports automatic detection for certain families of multitasking primitives only. Extend the support using this XML entry.

If your thread-creation function, for instance, does not belong to one of the supported families, map your function to a supported concurrency primitive.

See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments” on page 18-30.

Blocking Functions or Keywords

XML Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
  <functions>
    <function name="my_func">
      <behavior name="FORBIDDEN_FUNC"/>
    </function>
  </functions>
```

```

    </functions>
</specifications>

```

Use this entry in the XML file to specify that a function *my_func* must not be used in your source code.

XML Syntax:

```

<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
  <tokens>
    <token name="my_keyword" kind="keyword">
      <behavior name="FORBIDDEN_KEYWORD"/>
    </token>
  </tokens>
</specifications>

```

Use this entry in the XML file to specify that a keyword *my_keyword* must not be used in your source code.

See “Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers” on page 18-21.

Extending Initialization Checks

XML Syntax:

```

<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
  <functions>
    <function name="my_func">
      <check name="ARGUMENT_POINTS_TO_INITIALIZED_VALUE" arg="n"/>
    </function>
  </functions>
</specifications>

```

Use this entry in the XML file to specify if the pointer argument to a function *my_func* must point to an initialized buffer. The number *n* specifies which argument must be checked for buffer initialization.

See “Extend Checkers for Initialization to Check Function Arguments Passed by Pointers” on page 18-33.

Modifying Global Behavior

XML Syntax:

```

<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
  <global_scope>
    <parameter name="MAX_NUMBER_NESTED_LEVEL_CONTROL_FLOW" value="n1"/>
    <parameter name="MAX_NUMBER_NESTED_LEVEL_INCLUDES" value="n2"/>
    <parameter name="MAX_NUMBER_CONSTANT_IN_ENUMERATION" value="n3"/>
    <parameter name="MAX_NUMBER_MACROS_TRANSLATION_UNIT" value="n4"/>
    <parameter name="MAX_NUMBER_MEMBERS_IN_STRUCT" value="n5"/>
    <parameter name="MAX_NUMBER_NESTED_MEMBERS_IN_STRUCT" value="n6"/>
    <parameter name="NUMBER_SIGNIFICANT_CHARACTER_EXTERNAL_IDENTIFIER" value="n7"/>
    <parameter name="NUMBER_SIGNIFICANT_CHARACTER_INTERNAL_IDENTIFIER" value="n8"/>
  </global_scope>
</specifications>

```

```
</global_scope>
</specifications>
```

Here, $n1, \dots, n8$ specifies numerical values.

Use the entries $n1, \dots, n6$ for the following parameters to specify limits on certain aspects of your program. The modifications affect the checking of MISRA C:2012 Rule 1.1.

- `MAX_NUMBER_NESTED_LEVEL_CONTROL_FLOW`: Maximum depth of nesting allowed in control flow statements.
- `MAX_NUMBER_NESTED_LEVEL_INCLUDES`: Maximum levels of inclusion allowed using include files.
- `MAX_NUMBER_CONSTANT_IN_ENUMERATION`: Maximum number of constants allowed in an enumeration.
- `MAX_NUMBER_MACROS_TRANSLATION_UNIT`: Maximum number of macros allowed in a translation unit.
- `MAX_NUMBER_MEMBERS_IN_STRUCT`: Maximum number of members allowed in a structure.
- `MAX_NUMBER_NESTED_MEMBERS_IN_STRUCT`: Maximum levels of nesting allowed in a structure.

Use the entries $n7$ and $n8$ to specify how many characters must be compared to determine if two identifiers as identical. The modifications affect the checking of Rules 5.x.

- `NUMBER_SIGNIFICANT_CHARACTER_EXTERNAL_IDENTIFIER`: Number of characters to compare for external identifiers. External identifiers are ones declared with global scope or storage class `extern`.
- `NUMBER_SIGNIFICANT_CHARACTER_INTERNAL_IDENTIFIER`: Number of characters to compare for internal identifiers.

Specifying Real-Time Functions

XML Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
  <functions>
    <function name="my_func">
      <behavior name="REAL_TIME_FUNC"/>
    </function>
  </functions>
</specifications>
```

Use this entry in the XML file to specify a function `my_func` as a real-time function See AUTOSAR C++14 Rule A18-5-7.

Specifying Functions that Manage Memory

XML Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
  <functions>
    <function name="my_func">
      <behavior name="MANAGES_MEMORY"/>
    </function>
  </functions>
</specifications>
```

```

    </functions>
</specifications>

```

Use this entry in the XML file to specify a function *my_func* as a function that allocates or deallocates dynamic memory. See AUTOSAR C++14 Rule A18-5-7.

Specifying Functions that Handle Exceptions

XML Syntax:

```

<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
  <functions>
    <function name="my_func">
      <behavior name="EXCEPTION_HANDLING"/>
    </function>
  </functions>
</specifications>

```

Use this entry in the XML file to specify a function *my_func* as a function that handles one or more exceptions. See AUTOSAR C++14 Rule A15-0-7.

Specifying Critical Data Members

XML Syntax:

```

<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
  <members>
    <member name="my_critical_object" kind="variable">
      <behavior name="CRITICAL_DATA"/>
    </member>
  </members>
</specifications>

```

Use this entry in the XML file to specify the data member *my_critical_object* as critical. See Critical data member is not private.

Datalog Format

Datalog is a declarative logic programming language that Polyspace Bug Finder uses to create abstractions of function behavior. You can enter code behavior specifications in Datalog and add to the set of functions already abstracted for the Polyspace analysis.

The Datalog language, as used in Polyspace, is based on declaring relations between elements in the code. However, you do not need to know the full Datalog language to hook into the Polyspace analysis. You can simply invoke already declared relations from the Datalog-based API, using a function-like syntax.

Datalog Example

Consider the following example. The function `insertIntoDatabase()` reads a string from standard input and uses the string in the construction of an SQL query.

```

#include <stdio.h>
#include <string.h>

```



```

#include <sqlite3.h>

void addToName(char *, char *);

// Function to insert names into database
void insertIntoDatabase(void)
{
    sqlite3* db;
    char query[256] = "";
    char partialName[256];
    char fullName[256];

    char* error_message;
    int nameLength;

    sqlite3_open("userCredentials.db", &db);

    fread(partialName, 1, 128, stdin);

    addToName(partialName, fullName); //Function with unknown semantics

    strcat(query, "INSERT INTO privatenames (name) VALUES ('");
    strcat(query, fullName);
    strcat(query, "');");

    // Possible SQL injection here
    if (sqlite3_exec(db, query, NULL, NULL, &error_message))
    {
        sqlite3_free(error_message);
    }
}

```

This code has the possibility of an SQL injection if the function `addToName()` copies the content of its first argument to its second argument entirely or partly. The function is effectively passing on contents entered through standard input into the SQL query.

However, the defect checker `SQL injection` cannot find this defect because the function `addToName()` is not defined in this code and its semantics are not known. (In some cases, if the function is sufficiently complex, the analysis might lose track of a data flow from one function argument to another despite the function being defined.)

To help with the analysis, you can externally specify the semantics of the `addToName()` function using Datalog syntax. In a file with extension `.dl`, for instance `bufferFlows.dl`, enter the following:

```
.include "models/interfaces/taint.dl"
```

```
Taint.Basic.hasFlow("addToName", $InParameterDeref(0), $OutParameterDeref(1), "Source buffer copy")
```

This file:

- Includes the file `polyspaceroot\polyspace\pql\operating_system\models\interfaces\taint.dl`. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\R2023a` and `operating_system` is one of `win64`, `glxa64`, and `maci64`.

This file contains declarations of the relations that you invoke in your file.

- Invokes the relation `Taint.Basic.hasFlow` to specify that there is a data flow from the first parameter of the function `addToName()` to the second parameter:
 - `$InParameterDeref(0)` indicates the first parameter and designates the parameter as an input parameter. If a parameter is designated as an input parameter, its value prior to the function call is relevant for the relation.
 - `$OutParameterDeref(1)` indicates the second parameter and designates the parameter as output parameter. If a parameter is designated as an output parameter, its value after the function call is relevant for the relation.
 - The last argument is a message that can be associated with the function `addToName`. If Bug Finder finds a defect that uses the Datalog specification, the message appears in the event list below the defect.

Specify the Datalog file as argument for the analysis option `-code-behavior-specifications`, for instance:

```
-code-behavior-specifications bufferFlows.dl
```

The SQL injection checker is now able to detect the data flow through the function `addToName()` and report a possible SQL injection when the function `sqlite3_exec()` is executed.

Behaviors Supported by Datalog Format

The following checkers can be modified using code behavior specifications in Datalog format.

- SQL injection
- LDAP injection

The following code behaviors can be specified to modify the checkers:

- `Taint.Basic.hasFlow`: Specifies a function that propagates data from one function argument to another.

For instance, suppose that the following function `addToName()` propagates data from the dereference of the first parameter to the dereference of the second parameter:

```
void addToName (char *, char *);
```

You can specify this behavior using the following Datalog relation:

```
Taint.Basic.hasFlow("addToName", $InParameterDeref(0), $OutParameterDeref(1), "Source buffer c
```

All supported checkers can be modified using this code behavior.

- `Untrusted.Basic.isSource`: Specifies a function parameter or return value that represents potentially tainted data.

For instance, the following specifications:

```
Untrusted.Basic.isSource("fileOpen", $OutParameterDeref(1), "Opens an untrusted stream").
Untrusted.Basic.isSource("fileOpen_v2", $OutReturnDeref(), "Returns an untrusted stream").
```

Imply that these data are potentially tainted:

- Dereference of the second parameter of a `fileOpen()` function, for instance with signature `FILE * fileOpen (const char *, FILE * , const char*)`.

- Dereference of the return value of a `fileOpen_v2()` function, for instance with signature `FILE * fileOpen_v2 (int, const char*)`.

All supported checkers can be modified using this code behavior.

- `Sql.Basic.execution`: Specifies a function that executes an SQL query.

For instance, suppose that the following function `sql_exec()` executes an SQL query and the first parameter is the query string:

```
int sql_exec (char*, sqlite3* , int (*callback)(void*,int,char**,char**) , void*, char**);
```

You can specify this behavior using the following Datalog relation:

```
Sql.Basic.execution("sql_exec", $InParameterDeref(0)).
```

The checker that can be modified using this behavior is `SQL injection`.

- `Sql.Basic.sanitizing`: Specifies a function that sanitizes data to be used in an SQL query.

For instance, suppose that the following function `sql_sanitize()` sanitizes data to be used in an SQL query and the first parameter is the sanitized string:

```
int sql_sanitize (char *, int);
```

You can specify this behavior using the following Datalog relation:

```
Sql.Basic.sanitizing("sql_sanitize", $OutParameterDeref(0)).
```

The checker that can be modified using this behavior is `SQL injection`.

- `Ldap.Basic.sensitive`: Specifies a function that executes an LDAP query.

For instance, suppose that the following function `search_ldap_db()` executes an LDAP search and the fourth parameter is the search filter string:

```
int search_ldap_db (LDAP *, const char *, int, const char *, char **, int);
```

You can specify this behavior using the following Datalog relation:

```
Ldap.Basic.sensitive("search_ldap_db", $InParameterDeref(3)).
```

The checker that can be modified using this behavior is `LDAP injection`.

- `Ldap.Basic.sanitizing`: Specifies a function that sanitizes data to be used in an LDAP query.

For instance, suppose that the following function `ldap_validate_inputs()` sanitizes data to be used in an LDAP search and the first parameter is the sanitized string:

```
int ldap_validate_inputs (char *, int);
```

You can specify this behavior using the following Datalog relation:

```
Ldap.Basic.sanitizing("ldap_validate_inputs", $OutParameterDeref(0)).
```

The checker that can be modified using this behavior is `LDAP injection`.

See Also

-code-behavior-specifications

Related Examples

- “Modify Default Behavior of Bug Finder Checkers” on page 18-3

Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers

This topic shows how to create a custom list of forbidden functions, keywords or macros and check for use of these items in your code using Polyspace Bug Finder.

Identify Need for Extending Checker

Before creating or extending a checker, identify if an existing checker meets your requirements. For instance, these checkers flag the use of specific functions:

- **Use of dangerous standard function:** The checker flags functions that introduce the risk of buffer overflows and have safer alternatives.
- **Use of obsolete standard function:** The checker flags functions that are deprecated by the C/C++ standard.
- **Unsafe standard encryption function, Unsafe standard function:** The checkers flag functions that are unsafe to use in security-sensitive contexts.
- **Inefficient string length computation, `std::endl` may cause an unnecessary flush:** The checkers flag functions that can impact performance and have more efficient alternatives.

However, you might want to block functions that are not covered by an existing checker. For instance, you might want to forbid the use of signal handling functions such as `std::signal`:

```
#include <csignal>
#include <iostream>

namespace
{
    volatile std::sig_atomic_t gSignalStatus;
}

void signal_handler(int signal)
{
    gSignalStatus = signal;
}

int main()
{
    // Install a signal handler
    std::signal(SIGINT, signal_handler);

    std::cout << "SignalValue: " << gSignalStatus << '\n';
    std::cout << "Sending signal " << SIGINT << '\n';
    std::raise(SIGINT);
    std::cout << "SignalValue: " << gSignalStatus << '\n';
}
```

Likewise, you might want to block keywords that are not forbidden by an existing checker.

Extend Checker

If the functions, keywords or macros that you want to block are not covered by existing checkers, use one or more of these checkers:

- Use of a forbidden function
- Use of a forbidden keyword
- Use of a forbidden macro

To create a blocklist for the checker:

- 1 List functions, keywords and macros in an XML file in a specific syntax.

Copy the template file `code-behavior-specifications-template.xml` from the folder `polyspaceroot\polyspace\verifier\cxx` to a writable location and modify the file.

- Enter each forbidden function in the file using the following syntax after existing similar entries:

```
<function name="funcname">
  <behavior name="FORBIDDEN_FUNC"/>
</function>
```

where *funcname* is the name of the function you want to block. Remove previously existing entries in the file to avoid warnings.

- Enter each keyword in the file using the following syntax after existing similar entries:

```
<token name="keywordname" kind="keyword">
  <behavior name="FORBIDDEN_KEYWORD"/>
</token>
```

where *keywordname* is the name of the keyword you want to block.

- Enter each macro in the file using the following syntax after existing similar entries:

```
<token name="macroname" kind="macro">
  <behavior name="FORBIDDEN_MACRO"/>
</token>
```

where *macroname* is the name of the macro you want to block.

Note that you can use the * wildcard for functions and keywords (but not for macros). For instance, to forbid all functions whose names contain DEBUG, you can enter:

```
<function name="*DEBUG*">
  <behavior name="FORBIDDEN_FUNC"/>
</function>
```

- 2 Specify this XML file as argument for the option `-code-behavior-specifications`.

Checkers That Can Be Extended

The following checkers can be extended in this way:

- Use of a forbidden function
- Use of a forbidden keyword

- Use of a forbidden macro

See Also

-code-behavior-specifications|Use of a forbidden function|Use of a forbidden keyword|Use of a forbidden macro

More About

- “Modify Default Behavior of Bug Finder Checkers” on page 18-3

Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries

This topic shows how to create checkers for your custom library functions by mapping them to equivalent functions from the Standard Library.

Identify Need for Extending Checker

If you identify a Bug Finder checker that applies to a Standard Library function and can be extended to your custom library function, use this technique.

For instance, you might define a math function that has the same domain as a Standard Library math function. If Bug Finder checks for domain errors when using the Standard Library function, you can perform the same checks for the equivalent custom function.

Suppose that you define a function `acos32` that expects values in the range `[-1,1]`. You might want to detect if the function argument falls outside this range at run time, for instance, in this code snippet:

```
#include<math.h>
#include<float.h>

double acos32(double);
const int periodicity = 1.0;

int isItPeriodic() {
    return(abs(func(0.5) - func(0.5 + periodicity)) < DBL_MIN);
}

double func(double val) {
    return acos32(val);
}
```

One of the arguments to `acos32` is outside its allowed domain. If you do not provide the implementation of `acos32` or if the analysis of the `acos32` implementation is not precise, Bug Finder might not detect the issue. However, the function has the same domain as the Standard Library function `acos`. You can extend the checker `Invalid use of standard library floating point routine` that detects domain errors in uses of `acos` to detect the same kinds of errors with `acos32`.

If your custom function does not have a constrained domain but returns values in a constrained range, you can still map the function to an equivalent Standard Library function (if one exists) for more precise results on other checkers. For instance, you can map a function `cos32` that returns values in the range `[-1,1]` to the Standard Library function `cos`.

Extend Checker

You can extend checkers on functions from the Standard Library by mapping those functions to your custom library functions. For instance, in the preceding example, you can map the function `acos32` to the Standard Library function `acos`.

To perform the mapping:

- 1 List each mapping in an XML file in a specific syntax.

Copy the template file `code-behavior-specifications-template.xml` from the folder `polyspaceroot\polyspace\verifier\cxx` to a writable location and modify the file. Enter the mapping in the file using the following syntax after existing similar entries:

```
<function name="acos32" std="acos"> </function>
```

Remove previously existing entries in the file to avoid warnings.

- 2 Specify this XML file as argument for the option `-code-behavior-specifications`.

Checkers That Can Be Extended

The following checkers can be extended in this way:

- Invalid use of standard library floating point routine
- Invalid use of standard library integer routine
- AUTOSAR C++14 Rule A0-4-4
- CERT C: Rule FLP32-C
- CERT C++: FLP32-C
- MISRA C:2004 Coding Rule 20.3 on page 17-9

Limitations

You can map your custom function to a standard library function with similar semantics, subject to the following constraints:

- Your custom function must have the same number of arguments as the standard library function or more. Make sure that every argument of the standard library function is mapped to an argument of the custom function. For examples of argument remapping, see also `-code-behavior-specifications`.
- The mapped function arguments must have compatible data types. Likewise, the custom function must have a return type that is compatible with the standard library function. For instance:
 - An integer type (`char`, `int`, etc.) is not compatible with a floating point type (`float`, `double`, etc.)
 - A fundamental type is not compatible with a structure or enumeration.
 - A pointer type is not compatible with a non-pointer type.

See Also

`-code-behavior-specifications`

More About

- “Modify Default Behavior of Bug Finder Checkers” on page 18-3

Extend Bug Finder Checkers to Find Defects from Specific System Input Values

This topic shows how to find possible defects from specific values of system inputs. Unlike Code Prover, Bug Finder does not exhaustively check for run-time errors for all combinations of system inputs. However, you can extend some Bug Finder checkers and find if there are specific system input values that can lead to run-time errors.

Identify Need for Extending Checker

First identify if an existing checker is sufficient for your requirements.

For instance, the Bug Finder checker `Integer division by zero` detects if a division operation can have a zero denominator. Suppose, a library function has the possibility of a division by zero following several numerical operations. For instance, consider the function `speed` here:

```
#include <assert.h>

int speed(int k) {
    int i,j,v;
    i=2;
    j=k+5;
    while (i <10) {
        i++;
        j+=3;
    }

    v = 1 / (i-j);
    return v+k;
}
```

Suppose you see a sporadic run-time error when your program execution enters this function and the default Bug Finder analysis does not detect the issue. To minimize false positives, the default analysis might suppress issues from specific values of an unknown input (what if this value did not occur in practice at run time?). See also “Inputs in Polyspace Bug Finder”. To find the root cause of the sporadic error, you can run a stricter Bug Finder analysis for just this function.

Note that even after extending the checkers, Bug Finder does not provide the sound and exhaustive analysis of Code Prover. For instance, if Bug Finder does not detect errors after extending the checkers, this absence of detected errors does not have the same guarantees as green checks in Code Prover.

Extend Checker

To extend the checker and detect the above issue, use these options:

- Run stricter checks considering all values of system inputs (`-checks-using-system-input-values`): Enable this option. Checkers that rely on numerical values can now consider all input values for functions with at least one callee. You can change which functions are considered with the next option.
- Consider inputs to these functions (`-system-inputs-from`): Use the value `custom` and enter the name of the function whose inputs must be considered, in this case, `speed`. At the command line, use the option argument `custom=speed`.

When you run a Bug Finder analysis, you see a possible integer division by zero on the division operation. The result shows an example of an input value to the function `speed` that eventually leads to the current defect (zero value of the denominator).

Integer division by zero (Impact: High) ? ↻

Divisor is 0.

Result includes example values that lead to the defect.

	Event	File	Scope	Line
1	Function called by external code with input 'k' Possible input value causing defect: -19	bug.c	speed()	3
2	Entering function 'speed'	bug.c	speed()	3
3	Assignment to local variable 'i'	bug.c	speed()	5
4	Assignment to local variable 'j'	bug.c	speed()	6
5	Entering while loop	bug.c	speed()	7
6	Assignment to local variable 'i'	bug.c	speed()	8
7	Assignment to local variable 'j'	bug.c	speed()	9
8	Integer division by zero	bug.c	speed()	12

The tooltips on the defect show how the input value propagates through the code to eventually lead to one set of values that cause the defect.

```

1  #include <assert.h>
2
3  int speed(int k) {
4      int i,j,v;
5      i=2;
6      j=k+5;
7      while (i <=10) {
8          i++;
9          j+=3;
10     }
11
12     v = 1 / (i-j);
13     return v+k;
14 }

```

Checkers That Can Be Extended

The following checkers are affected by numerical values of inputs and can be extended using the preceding options:

- Bug Finder defects:

- Array access out of bounds
- Assertion
- Bitwise operation on negative value
- Float conversion overflow
- Float overflow
- Float division by zero
- Integer conversion overflow
- Integer division by zero
- Integer overflow
- Invalid use of standard library floating point routine
- Invalid use of standard library integer routine
- Non-initialized pointer
- Non-initialized variable
- Null pointer
- Pointer to non initialized value converted to const pointer
- Shift of a negative value
- Shift operation overflow
- Sign change integer conversion overflow
- Unsigned integer conversion overflow
- Unsigned integer overflow
- Use of plain char type for numeric value
- AUTOSAR C++14 coding rule:
 - AUTOSAR C++14 Rule A0-4-4
 - AUTOSAR C++14 Rule A3-8-1
 - AUTOSAR C++14 Rule A4-7-1
 - AUTOSAR C++14 Rule M5-0-16
 - AUTOSAR C++14 Rule A5-2-5
 - AUTOSAR C++14 Rule A5-3-2
 - AUTOSAR C++14 Rule A5-6-1
- CERT C rules and recommendations:
 - CERT C: Rule ARR30-C
 - CERT C: Rule EXP33-C
 - CERT C: Rule EXP34-C
 - CERT C: Rule FLP34-C
 - CERT C: Rule FLP32-C
 - CERT C: Rule INT30-C
 - CERT C: Rule INT31-C
 - CERT C: Rule INT32-C

- CERT C: Rule INT33-C
- CERT C: Rule INT34-C
- CERT C: Rec. EXP08-C
- CERT C: Rec. INT00-C
- CERT C: Rec. INT02-C
- CERT C: Rec. INT08-C
- CERT C: Rec. INT13-C
- CERT C: Rec. INT18-C
- CERT C: Rec. FLP03-C
- CERT C: Rec. FLP06-C
- CERT C++ rules:
 - CERT C++: ARR30-C
 - CERT C++: CTR50-CPP
 - CERT C++: EXP34-C
 - CERT C++: EXP53-CPP
 - CERT C++: EXP54-CPP
 - CERT C++: FLP32-C
 - CERT C++: FLP34-C
 - CERT C++: INT30-C
 - CERT C++: INT31-C
 - CERT C++: INT32-C
 - CERT C++: INT33-C
 - CERT C++: INT34-C
 - CERT C++: STR53-CPP
- MISRA C:2012 Rule 18.1
- MISRA C++:2008 Rule 5-0-16
- MISRA C:2004 Coding Rule 20.3 on page 17-9
- ISO/IEC TS 17961 rules:
 - ISO/IEC TS 17961 [diverr]
 - ISO/IEC TS 17961 [intoflow]
 - ISO/IEC TS 17961 [invptr]
 - ISO/IEC TS 17961 [nullref]
 - ISO/IEC TS 17961 [uninitref]

See Also

More About

- “Modify Default Behavior of Bug Finder Checkers” on page 18-3

Extend Concurrency Defect Checkers to Unsupported Multithreading Environments

This topic shows how to adapt concurrency defect checkers to unsupported multithreading environments, for instance, when a new thread creation is not detected automatically.

Identify Need for Extending Checker

By default, Bug Finder can detect concurrency primitives in certain families only (in Code Prover, the same automatic detection is available on an option). See “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 15-7. If you use primitives that do not belong to one of the supported families but have similar syntaxes, you can map your thread creation and other concurrency-related functions to the supported functions.

For instance, the following example uses:

- The function `createTask` to create a new thread.
- The function `takeLock` to begin a critical section.
- The function `releaseLock` to end the critical section.

```
typedef void* (*FUNT) (void*);

extern int takeLock(int* t);
extern int releaseLock(int* t);
// First argument is the function, second the id
extern int createTask(FUNT,int*,int*,void*);

int t_id1,t_id2;
int lock;

int var1;
int var2;

void* task1(void* a) {
    takeLock(&lock);
    var1++;
    var2++;
    releaseLock(&lock);
    return 0;
}

void* task2(void* a) {
    takeLock(&lock);
    var1++;
    releaseLock(&lock);
    var2++;
    return 0;
}

void main() {
    createTask(task1,&t_id1,0,0);
    createTask(task2,&t_id2,0,0);
}
```

Bug Finder does not detect the invocation of `createTask` as the creation of a new thread where control flow goes to the start function of the thread (first argument of `createTask`). The incorrect placement of the function `releaseLock` in `task2` and the possibility of a data race on the unprotected shared variable `var2` remains undetected.

However, the signature of `createTask`, `takeLock` and `releaseLock` are similar to the corresponding POSIX functions, `pthread_create`, `pthread_mutex_lock` and `pthread_mutex_unlock`. The order of arguments of these functions might be different from their POSIX equivalents.

Extend Checker

Since a POSIX thread creation can be detected automatically, map your thread creation and other concurrency-related functions to their POSIX equivalents. For instance, in the preceding example, perform the following mapping:

- `createTask` → `pthread_create`
- `takeLock` → `pthread_mutex_lock`
- `releaseLock` → `pthread_mutex_unlock`

To perform the mapping:

- 1 List each mapping in an XML file in a specific syntax.

Copy the template file `code-behavior-specifications-template.xml` from the folder `polyspaceroot\polyspace\verifier\cxx` to a writable location and modify the file. Enter each mapping in the file using the following syntax after existing similar entries:

```
<function name="createTask" std="pthread_create" >
  <mapping std_arg="1" arg="2"></mapping>
  <mapping std_arg="3" arg="1"></mapping>
  <mapping std_arg="2" arg="3"></mapping>
  <mapping std_arg="4" arg="4"></mapping>
</function>
<function name="takeLock" std="pthread_mutex_lock" >
</function>
<function name="releaseLock" std="pthread_mutex_unlock" >
</function>
```

Note that when mapping `createTask` to `pthread_create`, argument remapping is required, because the arguments do not correspond exactly. For instance, the thread start routine is the third argument of `pthread_create` but the first argument of `createTask`.

Remove previously existing entries in the file to avoid warnings.

- 2 Specify this XML file as argument for the option `-code-behavior-specifications`.

If you cannot perform a mapping to one of the supported families of concurrency primitives, you have to set up the multitasking analysis manually. See “Configuring Polyspace Multitasking Analysis Manually” on page 15-17.

Checkers That Can Be Extended

The concurrency defect checkers that can be extended in this way are:

- Bug Finder defects:
 - Data race
 - Double lock and Double unlock
 - Missing lock and Missing unlock
 - Deadlock
- CERT C rules and recommendation:
 - CERT C: Rule CON35-C
 - CERT C: Rule CON43-C
 - CERT C: Rule POS51-C
 - CERT C: Rec. CON01-C
- CERT C++ rules:
 - CERT C++: CON43-C
 - CERT C++: CON53-CPP
- MISRA C:2012 Dir 4.13

Limitations

You can map your custom thread creation and lock-unlock functions to similar standard library functions, subject to the following constraints:

- Your custom function must have the same number of arguments as the standard library function or more. Make sure that every argument of the standard library function is mapped to an argument of the custom function. For examples of argument remapping, see also `-code-behavior-specifications`.
- The mapped function arguments must have compatible data types. Likewise, the custom function must have a return type that is compatible with the standard library function. For instance:
 - An integer type (`char`, `int`, etc.) is not compatible with a floating point type (`float`, `double`, etc.)
 - A fundamental type is not compatible with a structure or enumeration.
 - A pointer type is not compatible with a non-pointer type.

See Also

`-code-behavior-specifications`

More About

- “Modify Default Behavior of Bug Finder Checkers” on page 18-3

Extend Checkers for Initialization to Check Function Arguments Passed by Pointers

This topic shows how to extend checkers for initialization to check function arguments passed by pointers. By default, Bug Finder does not check these arguments for initialization at the point of function call because you might perform the initialization in the function body. However, for specific functions, you can extend the checkers to check arguments passed by pointers for initialization at the point of function call.

Identify Need for Existing Checker

Suppose that you consider some function calls as part of the system boundary and you want to make sure that you pass initialized buffers across the boundary. For instance, the Run-Time environment or `Rte_` functions in AUTOSAR allow a software component to communicate with other software components. You might want to ensure that pointer arguments to these functions point to initialized buffers.

For instance, consider this code snippet:

```
extern void Rte_Write_int(unsigned int, int*);

void writeValueToAddress() {
    const unsigned int module_id = 0xfe;
    int x;
    Rte_Write_int(module_id, &x);
}
```

The argument `x` is passed by pointer to the `Rte_Write_int` function. Bug Finder does not check `x` for initialization at the point of function call. In the body of `Rte_Write_int`, if you attempt to read `x`, Bug Finder flags the non-initialized variable. However, you might not be able to provide the module containing the function body for analysis and might want to detect that `x` is non-initialized at the point of function call itself.

Extend Checker

You can specify that pointer arguments to some functions must point to initialized buffers. For instance, to specify that `Rte_Write_int` is one such function:

- 1 List the function in an XML file in a specific syntax.

Copy the template file `code-behavior-specifications-template.xml` from the folder `polyspaceroot\polyspace\verifier\cxx` to a writable location and modify the file. Enter the function in the file using the following syntax after existing similar entries:

```
<function name="Rte_Write_int">
  <check name="ARGUMENT_POINTS_TO_INITIALIZED_VALUE" arg="2"/>
</function>
```

This syntax indicates that Bug Finder must check the second argument of the `Rte_Write_int` function to determine if the argument points to an initialized buffer. Remove previously existing entries in the file to avoid warnings.

You can also use the wildcard `*` to cover a group of functions. To specify all functions beginning with `Rte_Write_`, enter:

```
<function name="Rte_Write_*)>  
  <check name="ARGUMENT_POINTS_TO_INITIALIZED_VALUE" arg="2"/>  
</function>
```

- 2 Specify this XML file as argument for the option `-code-behavior-specifications`.

If you rerun the analysis, you see a **Non-initialized variable** defect on `&x` when the function `Rte_Write_int` is called.

Checkers That Can Be Extended

These checkers are extended by using this option:

- Non-initialized variable
- AUTOSAR C++14 Rule A3-8-1
- CERT C: Rule EXP33-C
- CERT C++ rules:
 - CERT C++: EXP53-CPP
 - CERT C++: EXP54-CPP
- ISO/IEC TS 17961 [uninitref]

See Also

`-code-behavior-specifications`

More About

- “Modify Default Behavior of Bug Finder Checkers” on page 18-3

Extend Data Race Checkers to Atomic Operations

A data race might occur when multiple threads perform concurrent operations on a shared variable. When the operations are executed in one machine instruction, they are atomic. For instance, reading a shared variable of type `char` might take a single machine instruction. Such atomic operations can be performed concurrently without triggering a data race. By default, Polyspace Bug Finder assumes that certain operations are atomic and excludes them from data race checks. See “Define Atomic Operations in Multitasking Code” on page 15-25.

The Polyspace assumptions about the atomic nature of operations might not apply to your environment. If you are unsure whether an operation is atomic in your environment, extend the data race checkers to include these operations.

Identify Need for Extending Checker

Operations that take more than one machine cycle to execute are nonatomic operations. For instance, consider this operation:

```
MYREG = (u32dma0_chbit << 8UL) | u32dma0_chbit;
```

This operation takes more than one cycle to be performed and is therefore non-atomic. Such operations are checked for data race conditions when you activate data race checkers. If you are checking operations that take multiple instructions, use the default data race checkers, such as:

- Data race
- CERT C: Rule CON43-C
- CERT C++: CON43-C

Operations that take a single machine instruction to execute are assumed to be atomic. See “Define Atomic Operations in Multitasking Code” on page 15-25.

Because different machines have different word size, the Polyspace assumptions about the atomic nature of operations might not apply to your environment. For instance, consider the operation:

```
long long var = 0;
```

In target hardware where the size of a `long long` object is less than or equal to the word size, this operation is atomic. In target hardware where the size of a `long long` object is greater than the word size, this operation is not atomic. For instance, the preceding operation is atomic when `-target` is `x86_64`, but not atomic when `-target` is `i386`. If you are not sure whether the code executes in a `x86_64` machine or in an `i386` machine, extend the data race checker to include such operations.

Extend Checker

To include the assumed atomic operations when checking for data race violations, specify the option `Detect Data Race in Atomic Operations (-detect-atomic-data-race)`. Consider this code:

```
#include<stdio.h>
```

```
long var;
```

```

void begin_critical_section(void);
void end_critical_section(void);

void task1(void) {
    var = 1;
}

void task2(void) {
    int local_var;
    local_var = var;
    printf("%d", local_var);
}

void task3(void) {
    begin_critical_section();
    /* Operations in task3 */
    end_critical_section();
}

```

In this code, the write operation `var=1;` in task `task1` executes concurrently with the read operation `local_var=var;` in task `task2`. By default, Polyspace assumes that the target processor is `i386`, where these operations occur within a single machine instruction. These operations are excluded from a data race check.

Data race might still occur in these operations when the target processor is different from `i386`. To detect possible data races in this code, specify the option `Detect Data Race in Atomic Operations` (`-detect-atomic-data-race`). At the command line, use this command:

- Windows:

```

polyspace-bug-finder -checkers data_race -lang cpp ^
-entry-points task1,task2,task3 ^
-critical-section-begin begin_critical_section:CS1 ^
-critical-section-end end_critical_section:CS1 ^
-detect-atomic-data-race

```

- Linux:

```

polyspace-bug-finder -checkers data_race -lang cpp \
-entry-points task1,task2,task3 \
-critical-section-begin begin_critical_section:CS1 \
-critical-section-end end_critical_section:CS1 \
-detect-atomic-data-race

```

After specifying the option `-detect-atomic-data-race`, Polyspace flags the variable `var`.

Checkers That Can Be Extended

The concurrency defect checkers that you can extend in this way are:

- CERT C: Rule `CON43-C`
- CERT C++: `CON43-C`
- Data race

See Also

-detect-atomic-data-race

More About

- “Modify Default Behavior of Bug Finder Checkers” on page 18-3

Prepare Checkers Configuration for Polyspace Bug Finder Analysis

Before you incorporate Polyspace as a tool in the software development process of your organization, first decide how you plan on using Polyspace to improve your code. Choose which source components to analyze, which issues to check for, and so on. You can then prepare analysis configuration files that reflect your choices.

Broadly speaking, a Bug Finder analysis configuration consists of two parts:

- Build configuration including sources and target
- Checkers configuration

This topic describes a workflow for creating your checkers configuration in a typical deployment scenario. You can adapt this workflow to the specific requirements of your project or organization.

Identify Checkers to Enable

Suppose that you want to establish certain coding standards across your organization. You might follow one of several approaches:

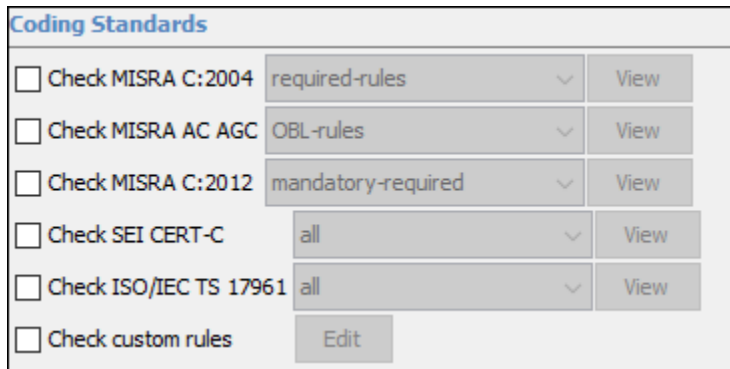
- Adhere to an external coding standard.

If Bug Finder supports the coding standard, you can select the standard and a predefined or custom set of rules from the standard.

Polyspace supports these external standards directly. For these standards, simply enable the standard in your configuration and start analysis.

- MISRA C:2004
- MISRA C:2012
- MISRA C++
- JSF AV C++
- AUTOSAR C++14 (*Bug Finder only*)
- CERT C (*Bug Finder only*)
- CERT C++ (*Bug Finder only*)
- ISO/IEC TS 17961 (*Bug Finder only*)
- Guidelines (*Bug Finder only*)

See “Check for and Review Coding Standard Violations” on page 16-2.



- Develop a set of in-house coding rules based on external standards and prior issues found.

See if you can automate checking of those rules through Bug Finder defect checkers and/or external coding standard checkers.

One way to locate a potential checker is to search by keywords in the documentation. Suppose you want to detect issues that can arise from use of variadic functions.

- 1 Search for keywords such as `variadic` or `va_arg` and refine search results by product to Bug Finder and then by category to **Review Analysis Results > Polyspace Bug Finder Results**.
- 2 Identify all checkers related to variadic functions. Note down the checkers that you want to enable. See if there is an overlap between checkers and eliminate duplicates.

You can record each defect checker that you enabled or disabled for your process requirements. You can start from the spreadsheet of checkers in `polyspaceroot\polyspace\resources\`. In the **Your Notes** column, note down your rationale for enabling or disabling a checker.

Defect Name	Your Notes
Incorrect data type passed to <code>va_arg</code>	Check that variadic function calls have no issues
Too many <code>va_arg</code> calls for current argument list	Check that variadic function calls have no issues

- Check only for defects (bugs) that are most likely to cause errors at run time.

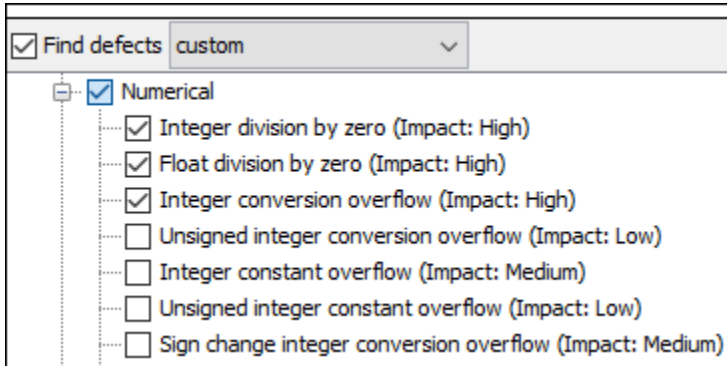
You might not be following standard coding practices in your organization and you might find external coding standards too sweeping for your preferences.

Start from the Bug Finder defect checkers and identify a subset of checkers for which you want to have zero unjustified defects. One way to identify this subset can be the following:

- First select defect checkers with high impact. These checkers can find issues that are likely to have serious consequences.

See also “Classification of Defects by Impact” on page 18-49.

- Run a first pass of Bug Finder analysis with high impact checkers and identify checkers that produce too much noise that you do not want to address immediately. You can disable these checkers for your initial deployment.



See also “Choose Specific Bug Finder Defect Checkers” on page 18-2.

You can follow a similar strategy with checkers for external coding standards. For instance, for MISRA C:2012, you can start from the mandatory or required guidelines and then choose to expand later.

At the end of this process, you have identified some checkers to enable in a Polyspace analysis. These checkers can be all defect (bug) checkers, or all checkers from external coding standards, or a mix of the two. The next section describes how to create checkers configuration files that you can deploy to your developers.

Create Checkers Configuration Files

A Polyspace Bug Finder analysis configuration is a list of analysis options specified using command-line flags. You can store the entire configuration in one options file, for instance, a text file named `allOptions.txt`, and specify the file using `-options-file` like this:

```
polyspace-bug-finder -options-file allOptions.txt
```

Or like this:

```
polyspace-bug-finder-server -options-file allOptions.txt
```

For your convenience, you can split the configuration into three parts:

- Build configuration (sources, targets, and so on).

Suppose that you save all options related to your build in a file `buildOptions.txt`. You can create this file manually or automatically from your build command (`makefile`).

For more information on how to create this file, see “Specify Target Environment and Compiler Behavior” on page 13-2.

- Defect checkers configuration.

Suppose that you specify defect checkers in a file `defectCheckers.txt`.

- External coding standard configuration.

Suppose that you specify a coding standard and associated checkers in a file `externalRuleCheckers.txt`.

You can string the files together in a run command like this:

```
polyspace-bug-finder
  -options-file buildOptions.txt
  -options-file defectCheckers.txt
  -options-file externalRuleCheckers.txt
```

This command combines the contents of all options files into one file. The splitting of one options file into several files has some advantages. By splitting into separate options files, you can, for instance, reuse the defect checkers configuration across projects while creating a build configuration individually for each project.

You have to then create the text files that specify the checkers that you choose to enable:

- The file `defectCheckers.txt` contains `-checkers` followed by a comma-separated list of the defect checkers that you choose to enable. For instance:

```
-checkers
  INT_ZERO_DIV,
  FLOAT_ZERO_DIV,
  ...
```

See also:

- `Find defects (-checkers)`
- “Short Names of Bug Finder Defect Groups and Defect Checkers” on page 30-11
- The file `externalRuleCheckers.txt` contains the coding standards that you want to enable and then refers to a separate XML file for specific rules from the standards.

For instance, a text file that enables specific rules from the MISRA C:2012 and AUTOSAR C++14 standard contains these options:

```
-misra3 from-file
-autosar-cpp14 from-file
-checkers-selection-file externalRuleCheckers.xml
```

The XML file `externalRuleCheckers.xml` that enables or disables checkers for rules from specific standards has this structure:

```

<polyspace_checkers_selection>
  <standard name="MISRA C:2004" state="off"/>
  <standard name="MISRA AC AGC" state="off"/>
  <standard name="MISRA C:2012" state="off"/>
  <standard name="MISRA C++:2008" state="off"/>
  <standard name="JSF AV C++" state="off"/>
  <standard name="SEI CERT C" state="off"/>
  <standard name="SEI CERT C++" state="off"/>
  <standard name="ISO/IEC TS 17961" state="off"/>
  <standard name="AUTOSAR C++14">
    <section name="0 Language independent issues">
      <check id="M0-1-1" state="on"/>
      <check id="M0-1-2" state="on"/>
      <check id="M0-1-3" state="off"/>
      <check id="M0-1-4" state="on">
        <comment>Not implemented</comment>
      </check>
      <check id="A0-1-1" state="on">
        <comment>Not implemented</comment>
      </check>
      <check id="A0-1-2" state="on"/>
      <check id="M0-1-8" state="on">
        <comment>Not implemented</comment>
      </check>
      .
      .
    </section>
  </standard>
</polyspace_checkers_selection>

```

For more information on how to create the XML file, see “Check for and Review Coding Standard Violations” on page 16-2.

You can create these files and use the final Polyspace run command in scripts. For instance:

- In a Jenkins build, you can specify the run command in a build script, along with other tools that you are running. After code submission, the Polyspace analysis can run on newly submitted code through the build scripts.
- In developer IDEs, you can specify the run command through a menu item that runs external tools. Developers can run the Polyspace analysis during coding by using the external tools.

Creating these options files by hand can be prone to errors. If you have a license of the desktop product, Polyspace Bug Finder, you can generate these files from the Polyspace user interface. See also “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 4-15.

See Also

More About

- “Choose Specific Bug Finder Defect Checkers” on page 18-2
- “Check for and Review Coding Standard Violations” on page 16-2

Bug Finder Defect Groups

In this section...
"C++ Exceptions" on page 18-43
"Concurrency" on page 18-43
"Cryptography" on page 18-44
"Data flow" on page 18-44
"Dynamic Memory" on page 18-45
"Good Practice" on page 18-45
"Numerical" on page 18-45
"Object Oriented" on page 18-45
"Performance" on page 18-46
"Programming" on page 18-46
"Resource Management" on page 18-47
"Static Memory" on page 18-47
"Security" on page 18-47
"Tainted data" on page 18-47

For convenience, the defect checkers in Bug Finder are classified into various groups.

- In certain projects, you can choose to focus only on specific groups of defects. Specify the group name for the option `Find defects (-checkers)`.
- When reviewing results, you can review all results of a certain group together. Filter out other results during review. See "Filter and Group Results in Polyspace Desktop User Interface" on page 23-2 or "Filter and Sort Results in Polyspace Access Web Interface" on page 27-8.

This topic gives an overview of the various groups.

C++ Exceptions

These defects are related to C++ exception handling. The defects include:

- Unhandled exception emitting from a `noexcept` function
- Unexpected exception arising during constructing the argument object of a `throw` statement
- `catch` statements catching exceptions by value instead of by reference
- `catch` statements hiding subsequent `catch` statements.

For more details about specific defects, see "C++ Exception Defects".

Command-Line Parameter: `cpp_exceptions`

Concurrency

These defects are related to multitasking code.

Data Race Defects

The data race defects occur when multiple tasks operate on a shared variable or call a nonreentrant standard library function without protection.

For the specific defects, see “Concurrency Defects”.

Command-Line Parameter: concurrency

Locking Defects

The locking defects occur when the critical sections are not set up appropriately. For example:

- The critical sections are involved in a deadlock.
- A lock function does not have the corresponding unlock function.
- A lock function is called twice without an intermediate call to an unlock function.

Critical sections protect shared variables from concurrent access. Polyspace expects critical sections to follow a certain format. The critical section must lie between a call to a lock function and a call to an unlock function.

For the specific defects, see “Concurrency Defects”.

Command-Line Parameter: concurrency

Cryptography

These defects are related to incorrect use of cryptography routines from the OpenSSL library. For instance:

- Use of cryptographically weak algorithms
- Absence of essential elements such as cipher key or initialization vector
- Wrong order of cryptographic operations

Note that these checkers support up to version 1.1.1 of the OpenSSL library.

For the specific defects, see “Cryptography Defects”.

Command-Line Parameter: cryptography

Data flow

These defects are errors relating to how information moves throughout your code. The defects include:

- Dead or unreachable code
- Unused code
- Non-initialized information

For the specific defects, see “Data Flow Defects”.

Command-Line Parameter: data_flow

Dynamic Memory

These defects are errors relating to memory usage when the memory is dynamically allocated. The defects include:

- Freeing dynamically allocated memory
- Unprotected memory allocations

For specific defects, see “Dynamic Memory Defects”.

Command-Line Parameter: `dynamic_memory`

Good Practice

These defects allow you to observe good coding practices. The defects by themselves might not cause a crash, but they sometimes highlight more serious logic errors in your code. The defects also make your code vulnerable to attacks and hard to maintain.

The defects include:

- Hard-coded constants such as buffer size and loop boundary
- Unused function parameters

For specific defects, see “Good Practice Defects”.

Command-Line Parameter: `good_practice`

Numerical

These defects are errors relating to variables in your code; their values, data types, and usage. The defects include:

- Mathematical operations
- Conversion overflow
- Operational overflow

For specific defects, see “Numerical Defects”.

Command-Line Parameter: `numerical`

Object Oriented

These defects are related to the object-oriented aspect of C++ programming. The defects highlight class design issues or issues in the inheritance hierarchy.

The defects include:

- Data member not initialized or incorrectly initialized in constructor
- Incorrect overriding of base class methods
- Breaking of data encapsulation

For specific defects, see “Object Oriented Defects”.

Command-Line Parameter: `object_oriented`

Performance

These defect checkers detect specific code patterns that directly cause or contribute to performance problems. Fixing the defects either directly removes performance problems, or removes contributing factors that might cause performance problems later.

These checkers can identify defects that a compiler's optimizations cannot fix. Compiler optimizations try to improve the performance of existing code, but must stay within language rules, and cannot make guesses about developer intent. On the contrary, these checkers can detect problems where:

- A more efficient code pattern can replace the current usage without changing functionalities, but the pattern is not required by the language rules.

For instance, the checker flags empty destructors in classes that do not have move constructors or move assignment operators. An empty destructor is allowed by the C++ language rules but prevents automatic generation of move operators. Removing the destructor allows compiler-generated destructors (and move operators) and makes the code more efficient without changing functionalities.

- Two or more similar implementations exist, and depending on requirements, the developer can choose a more efficient implementation.

For instance, the checker flags the use of `std::endl` in I/O operations. Both `std::endl` and `\n` introduce a new line, but if flushing is not intended, the latter is more efficient.

After these defects are fixed, the source code is more efficient or the developer intent is more explicit, and the compiler can more aggressively optimize the code.

The defects include:

- Issues that inadvertently cause copy instead of move operations
- Inefficient or unnecessary temporary variable creation
- Use of a function that has a possibly more efficient alternative

For specific defects, see "Performance Defects".

Command-Line Parameter: `performance`

Programming

These defects are errors relating to programming syntax. These defects include:

- Assignment versus equality operators
- Mismatches between variable qualifiers or declarations
- Badly formatted strings

For specific defects, see "Programming Defects".

Command-Line Parameter: `programming`

Resource Management

These defects are related to file handling. The defects include:

- Unclosed file stream
- Operations on a file stream after it is closed

For specific defects, see “Resource Management Defects”.

Command-Line Parameter: `resource_management`

Static Memory

These defects are errors relating to memory usage when the memory is statically allocated. The defects include:

- Accessing arrays outside their bounds
- Null pointers
- Casting of pointers

For specific defects, see “Static Memory Defects”.

Command-Line Parameter: `static_memory`

Security

These defects highlight places in your code which are vulnerable to hacking or other security attacks. Many of these defects do not cause runtime errors, but instead point out risky areas in your code. The defects include:

- Managing sensitive data
- Using dangerous or obsolete functions
- Generating random numbers
- Externally controlled paths and commands

For more details about specific defects, see “Security Defects”.

Command-Line Parameter: `security`

Tainted data

These defects highlight elements in your code which are from unsecured sources. Attackers can use input data or paths to attack your program and cause failures. These defects highlight elements in your code that are vulnerable. Defects include:

- Use of tainted variables or pointers
- Externally controlled paths

For more details about specific defects, see “Tainted Data Defects”. You can modify the behavior of the tainted data defects by using the optional command `-consider-analysis-perimeter-as-trust-boundary`. See `-consider-analysis-perimeter-as-trust-boundary`.

Command-Line Parameter: tainted_data

See Also

Find defects (-checkers)

More About

- “Short Names of Bug Finder Defect Groups and Defect Checkers” on page 30-11

Classification of Defects by Impact

To prioritize your review of Polyspace Bug Finder defects, you can use the **Impact** attribute assigned to the defect. The attribute allows you to filter results that require more immediate attention. For more information, see:

- “Filter and Group Results in Polyspace Desktop User Interface” on page 23-2
- “Filter and Sort Results in Polyspace Access Web Interface” on page 27-8

The **Impact** attribute is assigned to a defect based on the following considerations:

- Criticality, or whether the defect is likely to cause a code failure.

If a defect is likely to cause a code to fail, it is treated as a high impact defect. If the defect currently does not cause code failure but can cause problems with code maintenance in the future, it is a low impact defect.

- Certainty, or the rate of false positives.

For instance, the defect **Integer division by zero** is a high-impact defect because it is almost certain to cause a code crash. On the other hand, the defect **Dead code** has low impact because by itself, presence of dead code does not cause code failure. However, the dead code can hide other high-impact defects.

You cannot change the impact assigned to a defect.

High Impact Defects

The following list shows the high-impact defects.

C++ Exception

- Noexcept function exits with exception
- Throw argument raises unexpected exception
- Uncaught exception

Concurrency

- Data race
- Data race on adjacent bit fields
- Data race through standard library function call
- Deadlock
- Double lock
- Double unlock
- Missing unlock

Data Flow

- Non-initialized pointer
- Non-initialized variable

Dynamic Memory

- Deallocation of previously deallocated pointer
- Invalid deletion of pointer
- Invalid free of pointer
- Use of previously freed pointer

Numerical

- Absorption of float operand
- Float conversion overflow
- Float division by zero
- Integer conversion overflow
- Integer division by zero
- Invalid use of standard library floating point routine
- Invalid use of standard library integer routine

Object Oriented

- Base class assignment operator not called
- Copy constructor not called in initialization list
- Object slicing

Performance

- Invalid iterator usage

Programming

- Assertion
- Character value absorbed into EOF
- Declaration mismatch
- Errno not reset
- Incorrect value forwarding
- Invalid iterator usage
- Invalid use of == operator
- Invalid use of standard library routine
- Invalid va_list argument
- Misuse of errno
- Misuse of narrow or wide character string
- Misuse of return value from nonreentrant standard function
- Move operation on const object
- Non-compliance with AUTOSAR specification
- Possible misuse of sizeof
- Possibly unintended evaluation of expression because of operator precedence rules

- `std::string_view` initialized with dangling pointer
- Typedef mismatch
- Variable length array with nonpositive size
- Writing to `const` qualified object
- Wrong type used in `sizeof`

Resource Management

- Closing a previously closed resource
- Resource leak
- Use of previously closed resource
- Writing to read-only resource

Security

- Bad order of dropping privileges
- Critical data member is not private
- LDAP injection
- Privilege drop not verified
- Returned value of a sensitive function not checked
- SQL injection
- Unsafe call to a system function
- Use of non-secure temporary file

Static Memory

- Array access out of bounds
- Buffer overflow from incorrect string format specifier
- Destination buffer overflow in string manipulation
- Destination buffer underflow in string manipulation
- Invalid use of standard library memory routine
- Invalid use of standard library string routine
- Null pointer
- Pointer access out of bounds
- Pointer or reference to stack variable leaving scope
- Subtraction or comparison between pointers to different arrays
- Use of automatic variable as `putenv`-family function argument
- Use of path manipulation function without maximum sized buffer checking
- Wrong allocated object size for cast

Medium Impact Defects

The following list shows the medium-impact defects.

C++ Exception

- Exception caught by value
- Exception handler hidden by previous handler

Concurrency

- Asynchronously cancellable thread
- Atomic load and store sequence not atomic
- Atomic variable accessed twice in an expression
- Automatic or thread local variable escaping from a thread
- Destruction of locked mutex
- Join or detach of a joined or detached thread
- Missing lock
- Missing or double initialization of thread attribute
- Multiple mutexes used with same condition variable
- Thread-specific memory leak
- Use of undefined thread ID

Cryptography

- Constant block cipher initialization vector
- Constant cipher key
- Context initialized incorrectly for cryptographic operation
- Context initialized incorrectly for digest operation
- Incompatible padding for RSA algorithm operation
- Inconsistent cipher operations
- Incorrect key for cryptographic algorithm
- Missing blinding for RSA algorithm
- Missing block cipher initialization vector
- Missing certification authority list
- Missing cipher algorithm
- Missing cipher data to process
- Missing cipher final step
- Missing cipher key
- Missing data for encryption, decryption or signing operation
- Missing final step after hashing update operation
- Missing hash algorithm
- Missing padding for RSA algorithm
- Missing parameters for key generation
- Missing peer key
- Missing private key

- Missing private key for X.509 certificate
- Missing public key
- Missing salt for hashing operation
- Missing X.509 certificate
- No data added into context
- Nonsecure hash algorithm
- Nonsecure parameters for key generation
- Nonsecure RSA public exponent
- Nonsecure SSL/TLS protocol
- Predictable block cipher initialization vector
- Predictable cipher key
- Server certificate common name not checked
- TLS/SSL connection method not set
- TLS/SSL connection method set incorrectly
- Weak cipher algorithm
- Weak cipher mode
- Weak padding for RSA algorithm
- X.509 peer certificate not checked

Data Flow

- Pointer to non-initialized value converted to const pointer
- Unreachable code
- Useless if

Dynamic Memory

- Memory leak

Numerical

- Bitwise operation on negative value
- Integer constant overflow
- Integer overflow
- Sign change integer conversion overflow
- Use of plain char type for numerical value

Object Oriented

- Base class destructor not virtual
- Bytewise operations on nontrivial class object
- Conversion or deletion of incomplete class pointer
- Copy operation modifying source operand
- Incompatible types prevent overriding

- Member not initialized in constructor
- Missing virtual inheritance
- Operator new not overloaded for possibly overaligned class
- Partial override of overloaded virtual functions
- Return of non const handle to encapsulated data member
- Self assignment not tested in operator

Performance

- Const `std::move` input may cause a more expensive object copy
- Expensive allocation in loop
- Unnecessary use of `std::string::c_str()` or equivalent string methods
- Expensive constant `std::string` construction
- Expensive copy in a range-based for loop iteration
- Expensive local variable copy
- Expensive pass by value
- Expensive use of a standard algorithm when a more efficient method exists
- Expensive use of container's count method
- Expensive use of container's insertion method
- Expensive return by value
- Inefficient use of `sprintf`
- Missing call to container's reserve method
- Inefficient string length computation
- Move operation uses copy
- Missing `constexpr` specifier
- `std::endl` may cause an unnecessary flush
- Unnecessary padding
- `std::move` called on an unmovable type

Programming

- Abnormal termination of exit handler
- Bad file access mode or status
- Call through non-prototyped function pointer
- Copy of overlapping memory
- Environment pointer invalidated by previous operation
- Exception caught by value
- Exception handler hidden by previous handler
- Floating point comparison with equality operators
- Function called from signal handler not asynchronous-safe
- Function called from signal handler not asynchronous-safe (strict)
- Improper array initialization

- Improper erase-remove idiom
- Incorrect data type passed to `va_arg`
- Incorrect pointer scaling
- Incorrect type data passed to `va_start`
- Incorrect use of `offsetof` in C++
- Incorrect use of `va_start`
- Inline constraint not respected
- Invalid assumptions about memory organization
- Invalid file position
- Invalid use of `=` operator
- Memory comparison of padding data
- Memory comparison of strings
- Missing byte reordering when transferring data
- Misuse of `errno` in a signal handler
- Misuse of sign-extended character value
- Shared data access within signal handler
- Side effect in arguments to unsafe macro
- Signal call from within signal handler
- Standard function call with incorrect arguments
- Too many `va_arg` calls for current argument list
- Unnamed namespace in header file
- Unsafe conversion between pointer and integer
- Use of indeterminate string
- Use of `memset` with size argument zero

Resource Management

- Opening previously opened resource

Security

- Deterministic random output from constant seed
- `Errno` not checked
- Execution of a binary from a relative path can be controlled by an external actor
- File access between time of check and use (TOCTOU)
- File descriptor exposure to child process
- File manipulation after `chroot()` without `chdir("/")`
- Hard-coded sensitive data
- Inappropriate I/O operation on device files
- Incorrect order of network connection operations
- Load of library from a relative path can be controlled by an external actor

- Mismatch between data length and size
- Misuse of readlink()
- Predictable random output from predictable seed
- Sensitive data printed out
- Sensitive heap memory not cleared before release
- Uncertain memory cleaning
- Uncleared sensitive data in stack
- Unsafe standard encryption function
- Unsafe standard function
- Vulnerable permission assignments
- Vulnerable pseudo-random number generator

Static Memory

- Unreliable cast of function pointer
- Unreliable cast of pointer

Tainted Data

- Array access with tainted index
- Command executed from externally controlled path
- Execution of externally controlled command
- Host change using externally controlled elements
- Library loaded from externally controlled path
- Loop bounded with tainted value
- Memory allocation with tainted size
- Tainted sign change conversion
- Tainted size of variable length array
- Use of externally controlled environment variable

Low Impact Defects

The following list shows the low-impact defects.

Concurrency

- Blocking operation while holding lock
- Function that can spuriously fail not wrapped in loop
- Function that can spuriously wake up not wrapped in loop
- Multiple threads waiting on same condition variable
- Signal call in multithreaded program
- Use of signal to kill thread

Data Flow

- Code deactivated by constant false condition
- Dead code
- Missing return statement
- Partially accessed array
- Static uncalled function
- Useless preprocessor conditional directive
- Variable shadowing
- Write without a further read

Dynamic Memory

- Alignment changed after memory reallocation
- Mismatched alloc/dealloc functions on Windows
- Unprotected dynamic memory allocation

Good Practice

- Ambiguous declaration syntax
- Bitwise and arithmetic operation on the same data
- C++ reference to const-qualified type with subsequent modification
- C++ reference type qualified with const or volatile
- Declaration of catch for generic exception
- Declaration of throw for generic exception
- Delete of void pointer
- Duplicated code
- File does not compile
- Hard-coded buffer size
- Hard-coded loop boundary
- Hard-coded object size used to manipulate memory
- Incorrect syntax of flexible array member size
- Incorrectly indented statement
- Invalid scientific notation format
- Line with more than one statement
- Macro terminated with a semicolon
- Macro with multiple statements
- Method not const
- Missing break of switch case
- Missing overload of allocation or deallocation function
- Missing reset of a freed pointer
- Partially duplicated code

- Possibly inappropriate data type for switch expression
- Public static field is not const
- Redundant expression in sizeof operand
- Reference to un-named temporary
- Semicolon on same line as if, for or while statement
- Unmodified variable not const-qualified
- Unused parameter
- Use of a forbidden keyword
- Use of a forbidden macro
- Use of a forbidden function
- Use of setjmp/longjmp
- Useless Include

Numerical

- Float overflow
- Integer precision exceeded
- Possible invalid operation on boolean operand
- Precision loss in integer to float conversion
- Shift of a negative value
- Shift operation overflow
- Unsigned integer constant overflow
- Unsigned integer conversion overflow
- Unsigned integer overflow

Object Oriented

- *this not returned in copy assignment operator
- Lambda used as typeid operand
- Missing explicit keyword

Performance

- A move operation may throw
- Const parameter values may cause unnecessary data copies
- Const return values may cause unnecessary data copies
- Const rvalue reference parameter may cause unnecessary data copies
- Empty destructors may cause unnecessary data copies
- Expensive dynamic cast
- Expensive logical operation
- Expensive post-increment operation
- Expensive return caused by unnecessary std::move
- Expensive return of a const object

- Expensive use of container's size method
- Expensive use of map's bracket operator to insert or assign a value
- Expensive use of string function
- Expensive use of substr() to shorten a std::string
- Expensive use of non-member std::string operator+() instead of a simple append
- Expensive use of std::string methods instead of more efficient overload
- Expensive use of std::string with empty string literal
- Inefficient string length computation
- Inefficient use of for loop
- std::endl may cause an unnecessary flush
- Unnecessary construction before reassignment
- Unnecessary implementation of a special member function
- Use of new or make_unique instead of more efficient make_shared

Programming

- Accessing object with temporary lifetime
- Alternating input and output from a stream without flush or positioning call
- C string from string::c_str() compared to pointer
- Call to memset with unintended value
- Format string specifiers and arguments mismatch
- Memory comparison of float-point values
- Missing null in string array
- Misuse of a FILE object
- Misuse of structure with flexible array member
- Modification of internal buffer returned from nonreentrant standard function
- Overlapping assignment
- Possible copy-paste error
- Predefined macro used as an object
- Preprocessor directive in macro argument
- Qualifier removed in conversion
- Return from computational exception signal handler
- Side effect of expression ignored
- Stream argument with possibly unintended side effects
- Universal character name from token concatenation
- Unsafe conversion from string to numerical value

Security

- Function pointer assigned with absolute address
- Information leak via structure padding
- Missing case for switch condition
- Umask used with chmod-style arguments
- Use of dangerous standard function
- Use of obsolete standard function
- Vulnerable path manipulation

Static Memory

- Arithmetic operation with NULL pointer

Tainted Data

- Pointer dereference with tainted offset
- Tainted division operand
- Tainted modulo operand
- Tainted NULL or non-null-terminated string
- Tainted string format
- Use of tainted pointer

See Also**More About**

- “Filter and Group Results in Polyspace Desktop User Interface” on page 23-2

Sources of Tainting in a Polyspace Analysis

Generally, any code element that can be modified from outside of the code is considered tainted data. An attacker might pass values to tainted variables to cause program failure, inject malicious code, or leak resources. The results of operations that use tainted data are also considered tainted.. For instance, if you calculate a path to a file by using tainted variable, the file also becomes tainted. To mitigate risks associated with tainted data, validate the content of the data before you use it.

Enhance the security of your code by using the Polyspace tainted data defect checkers to identify sources of tainted data and then validating data from those sources.

Sources of Tainted Data

Polyspace considers data from these sources as tainted data:

- Volatile objects: Objects declared by using the keyword `volatile` can be modified by the hardware during program execution. Using volatile objects without checking their content might lead to segmentation errors, memory leak or security threat. Polyspace flags operations that use volatile objects without validating them.
- Functions that obtains a user input: Library functions such as `getenv`, `gets`, `read`, `scanf`, or `fopen` return user inputs such as an environment variable, a string, a data stream, formatted data or a file. The `main()` might also take input arguments directly from the user. User dependent inputs are unpredictable. Before using these input, validate them by checking their format, length, or content.
- Functions that interacts with hardware: Library functions such as `RegQueryValueEx` interacts with hardware like registers and peripherals. These functions return hardware dependent data that might be unpredictable. Before using data obtained from hardware, validate them by checking their format, length, or content.
- Functions that returns the current time: Library functions such as `ctime` returns the current time of the system as a formatted string. The format of the string depends on the environment. Before using such strings, validate them by checking their format.
- Functions that return a random number: Before using random numbers, validate them by checking their format and range.

To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`. See “Modify Default Behavior of Bug Finder Checkers” on page 18-3

Impact of Tainted Data Defects

An attacker might exploit tainted data defects by deliberately feeding unexpected input to the program to expose the stack or execute commands that access or delete sensitive data. Consider this code which uses input from the user to modify the system.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 128
void Echo(char* string, int n) {
    printf("Argument %d is; ",n);
    printf(string); //Tainted operation
}
```

```

void SystemCaller(char* string){
    printf("Calling System...");
    char cmd[MAX] = "/usr/bin/cat ";
    strcat(cmd, string);
    system(cmd); //Tainted operation
}

int main(int argc, char** argv) {
    int i = 0;
    for(i = 0; i < argc; ++i){
        Echo(argv[i], i);
        SystemCaller(argv[i]);
    }
    return (0);
}

```

The input from the user is tainted. Polyspace flags two tainted data defects in this code.

- In the function `Echo`, the line `printf(string)` print a user input string without validating the string. This defect enables an attacker to expose the stack by manipulating the input string. For instance, if the user input is `"%d"`, function prints the integer in the stack after `n` is printed.
- In the function `SystemCaller`, a user input string is used to call an operating system command. Malicious users can execute commands to access or delete sensitive data, and even crash the system by exploiting this defect.

To prevent such attacks, validate the tainted data by checking their format, length, or content. For instance, in this code, the tainted inputs are validated before they are used.

```

#include <stdio.h>
#include <stdlib.h>
#define MAX 128
extern char** LIST_OF_COMMANDS;
int isAllowed(char*);
void Echo(char* string, int n) {
    printf("Argument %d is; ", n);
    printf("%s", string); //Validated
}
void SystemCaller(char* string){
    printf("Calling System...");
    char cmd[MAX] = "/usr/bin/cat ";
    if(isAllowed(string)==1){
        strcat(cmd, string);
        system(cmd); //Validated
    }
}

int main(int argc, char** argv) {
    int i = 0;
    for(i = 0; i < argc || i < 10; ++i){
        Echo(argv[i], i);
        SystemCaller(argv[i]);
    }
    return (0);
}

```

By specifying the format as `%s` in `printf`, the tainted input string is validated. Now, the program prints the content of the string and the stack is no longer exposed. In `SystemCaller`, the program executes an operating system command only if the input matches an allowed command.

For details about the tainted data defects in Polyspace, see “Tainted Data Defects”.

Polyspace Tainted Data Checkers

Check Tainted Data by using these Bug Finder defects and coding rules:

- Bug Finder Defects: “Tainted Data Defects”
- AUTOSAR C++14 coding rules:
 - AUTOSAR C++14 Rule A27-0-1
 - AUTOSAR C++14 Rule A5-2-5
 - AUTOSAR C++14 Rule A5-6-1
- CERT C coding rules:
 - CERT C: Rule ARR30-C
 - CERT C: Rule ARR32-C
 - CERT C: Rec. ENV01-C
 - CERT C: Rule FI030-C
 - CERT C: Rec. INT04-C
 - CERT C: Rec. INT10-C
 - CERT C: Rule INT31-C
 - CERT C: Rule INT32-C
 - CERT C: Rule INT33-C
 - CERT C: Rec. MEM04-C
 - CERT C: Rec. MEM05-C
 - CERT C: Rule MEM35-C
 - CERT C: Rec. MSC21-C
 - CERT C: Rec. STR02-C
 - CERT C: Rule STR32-C
 - CERT C: Rec. WIN00-C
- CERT C++ Coding Rules:
 - CERT C++: ARR30-C
 - CERT C++: CTR50-CPP
 - CERT C++: FI030-C
 - CERT C++: INT31-C
 - CERT C++: INT32-C
 - CERT C++: INT33-C
 - CERT C++: MEM35-C
 - CERT C++: STR32-C

- CERT C++: STR53-CPP
- ISO/IEC TS 17961 rules:
 - ISO/IEC TS 17961 [taintformatio]
 - ISO/IEC TS 17961 [taintsink]
 - ISO/IEC TS 17961 [taintstrcpy]
 - ISO/IEC TS 17961 [usrfmt]

See Also

Find defects (-checkers) | -consider-analysis-perimeter-as-trust-boundary

More About

- “Tainted data” on page 18-47
- “Modify Default Behavior of Bug Finder Checkers” on page 18-3

Polyspace Bug Finder Defects Checkers Enabled by Default

When you start a Bug Finder analysis, these checkers are enabled by default. To view a list of checkers that are enabled by default when you analyze generated code, see “Polyspace Bug Finder Defects Checkers Enabled by Default for Generated Code” on page 18-70.

Defect	Command-line Name
Absorption of float operand	FLOAT_ABSORPTION
Accessing object with temporary lifetime	TEMP_OBJECT_ACCESS
Alignment changed after memory reallocation	ALIGNMENT_CHANGE
Alternating input and output from a stream without flush or positioning call	IO_INTERLEAVING
Array access out of bounds	OUT_BOUND_ARRAY
Assertion	ASSERT
Atomic load and store sequence not atomic	ATOMIC_VAR_SEQUENCE_NOT_ATOMIC
Atomic variable accessed twice in an expression	ATOMIC_VAR_ACCESS_TWICE
Base class assignment operator not called	MISSING_BASE_ASSIGN_OP_CALL
Base class destructor not virtual	DTOR_NOT_VIRTUAL
Buffer overflow from incorrect string format specifier	STR_FORMAT_BUFFER_OVERFLOW
Call through non-prototyped function pointer	UNPROTOTYPED_FUNC_CALL
Character value absorbed into EOF	CHAR_EOF_CONFUSED
Closing a previously closed resource	DOUBLE_RESOURCE_CLOSE
Conversion or deletion of incomplete class pointer	INCOMPLETE_CLASS_PTR
Copy constructor not called in initialization list	MISSING_COPY_CTOR_CALL
Copy operation modifying source operand	COPY_MODIFYING_SOURCE
Data race	DATA_RACE
Data race on adjacent bit fields	DATA_RACE_BIT_FIELDS
Data race through standard library function call	DATA_RACE_STD_LIB
Dead code	DEAD_CODE
Deadlock	DEADLOCK

Defect	Command-line Name
Deallocation of previously deallocated pointer	DOUBLE_DEALLOCATION
Declaration mismatch	DECL_MISMATCH
Destination buffer overflow in string manipulation	STRLIB_BUFFER_OVERFLOW
Destination buffer underflow in string manipulation	STRLIB_BUFFER_UNDERFLOW
Double lock	DOUBLE_LOCK
Double unlock	DOUBLE_UNLOCK
Environment pointer invalidated by previous operation	INVALID_ENV_POINTER
Errno not reset	MISSING_ERRNO_RESET
Exception caught by value	EXCP_CAUGHT_BY_VALUE
Exception handler hidden by previous handler	EXCP_HANDLER_HIDDEN
Float conversion overflow	FLOAT_CONV_OVFL
Float division by zero	FLOAT_ZERO_DIV
Format string specifiers and arguments mismatch	STRING_FORMAT
Improper array initialization	IMPROPER_ARRAY_INIT
Incompatible types prevent overriding	VIRTUAL_FUNC_HIDING
Incorrect data type passed to va_arg	VA_ARG_INCORRECT_TYPE
Incorrect pointer scaling	BAD_PTR_SCALING
Incorrect type data passed to va_start	VA_START_INCORRECT_TYPE
Incorrect use of offsetof in C++	OFFSETOF_MISUSE
Incorrect use of va_start	VA_START_MISUSE
Incorrect value forwarding	INCORRECT_VALUE_FORWARDING
Infinite loop	INFINITE_LOOP
Inline constraint not respected	INLINE_CONSTRAINT_NOT_RESPECTED
Integer conversion overflow	INT_CONV_OVFL
Integer division by zero	INT_ZERO_DIV
Invalid assumptions about memory organization	INVALID_MEMORY_ASSUMPTION
Invalid deletion of pointer	BAD_DELETE
Invalid free of pointer	BAD_FREE
Invalid use of = operator	BAD_EQUAL_USE
Invalid use of == operator	BAD_EQUAL_EQUAL_USE
Invalid use of standard library floating point routine	FLOAT_STD_LIB

Defect	Command-line Name
Invalid use of standard library integer routine	INT_STD_LIB
Invalid use of standard library memory routine	MEM_STD_LIB
Invalid use of standard library routine	OTHER_STD_LIB
Invalid use of standard library string routine	STR_STD_LIB
Invalid va_list argument	INVALID_VA_LIST_ARG
Lambda used as typeid operand	LAMBDA_TYPE_MISUSE
Memory comparison of padding data	MEMCMP_PADDING_DATA
Memory comparison of strings	MEMCMP_STRINGS
Missing lock	BAD_UNLOCK
Missing null in string array	MISSING_NULL_CHAR
Missing return statement	MISSING_RETURN
Missing unlock	BAD_LOCK
Misuse of a FILE object	FILE_OBJECT_MISUSE
Misuse of errno	ERRNO_MISUSE
Misuse of errno in a signal handler	SIG_HANDLER_ERRNO_MISUSE
Misuse of sign-extended character value	CHARACTER_MISUSE
Misuse of structure with flexible array member	FLEXIBLE_ARRAY_MEMBER_STRUCT_MISUSE
Move operation on const object	MOVE_CONST_OBJECT
Noexcept function exits with exception	NOEXCEPT_FUNCTION_THROWS
Non-initialized pointer	NON_INIT_PTR
Non-initialized variable	NON_INIT_VAR
Null pointer	NULL_PTR
Object slicing	OBJECT_SLICING
Opening previously opened resource	DOUBLE_RESOURCE_OPEN
Operator new not overloaded for possibly overaligned class	MISSING_OVERLOAD_NEW_FOR_ALIGNED_OBJ
Partial override of overloaded virtual functions	PARTIAL_OVERRIDE
Partially accessed array	PARTIALLY_ACCESSED_ARRAY
Pointer access out of bounds	OUT_BOUND_PTR
Pointer or reference to stack variable leaving scope	LOCAL_ADDR_ESCAPE
Possible misuse of sizeof	SIZEOF_MISUSE

Defect	Command-line Name
Possibly unintended evaluation of expression because of operator precedence rules	OPERATOR_PRECEDENCE
Predefined macro used as an object	MACRO_USED_AS_OBJECT
Preprocessor directive in macro argument	PRE_DIRECTIVE_MACRO_ARG
Resource leak	RESOURCE_LEAK
Return from computational exception signal handler	SIG_HANDLER_COMP_EXCP_RETURN
Self assignment not tested in operator	MISSING_SELF_ASSIGN_TEST
Shared data access within signal handler	SIG_HANDLER_SHARED_OBJECT
Side effect of expression ignored	SIDE_EFFECT_IGNORED
Sign change integer conversion overflow	SIGN_CHANGE
Signal call from within signal handler	SIG_HANDLER_CALLING_SIGNAL
Standard function call with incorrect arguments	STD_FUNC_ARG_MISMATCH
Stream argument with possibly unintended side effects	STREAM_WITH_SIDE_EFFECT
Subtraction or comparison between pointers to different arrays	PTR_TO_DIFF_ARRAY
Throw argument raises unexpected exception	THROW_ARGUMENT_EXPRESSION_THROWS
Too many va_arg calls for current argument list	TOO_MANY_VA_ARG_CALLS
Typedef mismatch	TYPEDEF_MISMATCH
Universal character name from token concatenation	PRE_UCNAME_JOIN_TOKENS
Unnamed namespace in header file	UNNAMED_NAMESPACE_IN_HEADER
Unreachable code	UNREACHABLE
Unreliable cast of function pointer	FUNC_CAST
Unreliable cast of pointer	PTR_CAST
Unsigned integer conversion overflow	UINT_CONV_OVFL
Use of automatic variable as putenv-family function argument	PUTENV_AUTO_VAR
Use of previously closed resource	CLOSED_RESOURCE_USE
Use of previously freed pointer	FREED_PTR
Useless if	USELESS_IF

Defect	Command-line Name
Variable length array with nonpositive size	NON_POSITIVE_VLA_SIZE
Variable shadowing	VAR_SHADOWING
Write without a further read	USELESS_WRITE
Writing to const qualified object	CONSTANT_OBJECT_WRITE
Writing to read-only resource	READ_ONLY_RESOURCE_WRITE
Wrong type used in sizeof	PTR_SIZEOF_MISMATCH

To enable other checkers and coding rule, configure checkers selections. See “Prepare Checkers Configuration for Polyspace Bug Finder Analysis” on page 18-38 or “Setting Checkers in Polyspace as You Code”.

See Also

More About

- “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78
- “Polyspace Bug Finder Defects Checkers Enabled by Default for Generated Code” on page 18-70

Polyspace Bug Finder Defects Checkers Enabled by Default for Generated Code

When you start a Bug Finder analysis on code generated from Simulink or MATLAB code, these checkers are enabled by default. To see a list of checkers that are enabled when you run an analysis on handwritten code, see “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 18-65.

Defect	Command-line Name
Absorption of float operand	FLOAT_ABSORPTION
Array access out of bounds	OUT_BOUND_ARRAY
Assertion	ASSERT
Data race	DATA_RACE
Data race on adjacent bit fields	DATA_RACE_BIT_FIELDS
Data race through standard library function call	DATA_RACE_STD_LIB
Dead code	DEAD_CODE
Deadlock	DEADLOCK
Deallocation of previously deallocated pointer	DOUBLE_DEALLOCATION
Declaration mismatch	DECL_MISMATCH
Dereference of a null pointer	NULL_PTR
Double lock	DOUBLE_LOCK
Double unlock	DOUBLE_UNLOCK
Float conversion overflow	FLOAT_CONV_OVFL
Float division by zero	FLOAT_ZERO_DIV
Format string specifiers and arguments mismatch	STRING_FORMAT
Infinite loop	INFINITE_LOOP
Integer conversion overflow	INT_CONV_OVFL
Integer division by zero	INT_ZERO_DIV
Invalid free of pointer	BAD_FREE
Invalid use of standard library floating point routine	FLOAT_STD_LIB
Invalid use of standard library integer routine	INT_STD_LIB
Invalid use of standard library memory routine	MEM_STD_LIB
Invalid use of standard library routine	OTHER_STD_LIB

Defect	Command-line Name
Invalid use of standard library string routine	STR_STD_LIB
Missing lock	BAD_UNLOCK
Missing return statement	MISSING_RETURN
Missing unlock	BAD_LOCK
Non-initialized pointer	NON_INIT_PTR
Non-initialized variable	NON_INIT_VAR
Pointer access out of bounds	OUT_BOUND_PTR
Sign change integer conversion overflow	SIGN_CHANGE
Typedef mismatch	TYPDEF_MISMATCH
Unreachable code	UNREACHABLE
Unreliable cast of function pointer	FUNC_CAST
Unreliable cast of pointer	PTR_CAST
Unsigned integer conversion overflow	UINT_CONV_OVFL
Use of previously freed pointer	FREED_PTR
Useless if	USELESS_IF

To enable other checkers and coding rule, configure checkers selections. See “Prepare Checkers Configuration for Polyspace Bug Finder Analysis” on page 18-38.

See Also

More About

- “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 18-65
- “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78

Bug Finder Results Found in Fast Analysis Mode

In fast analysis mode, Bug Finder checks for a subset of defects and coding rules only. The tables below list the results that can be found in a fast analysis. See also `Use fast analysis mode for Bug Finder (-fast-analysis)`.

These defects and coding standard violations are either found earlier in the analysis or leverage archived information from a previous analysis. The analysis results are comparatively easier to review and fix because most results can be understood by focusing on two or three lines of code (the line with the defect and one or two previous lines).

Because of the simplified nature of the analysis, you might see fewer defects in the fast analysis mode compared to a regular Bug Finder analysis.

Polyspace Bug Finder Defects

Static Memory

Name	Description
Buffer overflow from incorrect string format specifier (<code>str_format_buffer_overflow</code>)	String format specifier causes buffer argument of standard library functions to overflow
Unreliable cast of function pointer (<code>func_cast</code>)	Function pointer cast to another function pointer with different argument or return type
Unreliable cast of pointer (<code>ptr_cast</code>)	Pointer implicitly cast to different data type

Programming

Name	Description
Copy of overlapping memory (overlapping_copy)	Source and destination arguments of a copy function have overlapping memory
Exception caught by value (excp_caught_by_value)	catch statement accepts an object by value
Exception handler hidden by previous handler (excp_handler_hidden)	catch statement is not reached because of an earlier catch statement for the same exception
Format string specifiers and arguments mismatch (string_format)	String specifiers do not match corresponding arguments
Improper array initialization (improper_array_init)	Incorrect array initialization when using initializers
Invalid use of == operator (bad_equal_equal_use)	Equality operation in assignment statement
Invalid use of = operator (bad_equal_use)	Assignment in conditional statement
Invalid use of floating point operation (bad_float_op)	Imprecise comparison of floating point variables
Missing null in string array (missing_null_char)	String does not terminate with null character
Overlapping assignment (overlapping_assign)	Memory overlap between left and right sides of an assignment
Possibly unintended evaluation of expression because of operator precedence rules (operator_precedence)	Operator precedence rules cause unexpected evaluation order in arithmetic expression
Unsafe conversion between pointer and integer (bad_int_ptr_cast)	Misaligned or invalid results from conversions between pointer and integer types
Wrong type used in sizeof (ptr_sizeof_mismatch)	sizeof argument does not match pointed type

Data Flow

Name	Description
Code deactivated by constant false condition (deactivated_code)	Code segment deactivated by #if 0 directive or if(0) condition
Missing return statement (missing_return)	Function does not return value though return type is not void
Static uncalled function (uncalled_func)	Function with static scope not called in file
Variable shadowing (var_shadowing)	Variable hides another variable of same name with nested scope

Object Oriented

Name	Description
*this not returned in copy assignment operator (return_not_ref_to_this)	operator= method does not return a pointer to the current object
Base class assignment operator not called (missing_base_assign_op_call)	Copy assignment operator does not call copy assignment operators of base subobjects
Base class destructor not virtual (dtor_not_virtual)	Class cannot behave polymorphically for deletion of derived class objects
Copy constructor not called in initialization list (missing_copy_ctor_call)	Copy constructor does not call copy constructors of some members or base classes
Incompatible types prevent overriding (virtual_func_hiding)	Derived class method hides a virtual base class method instead of overriding it
Member not initialized in constructor (non_init_member)	Constructor does not initialize some members of a class
Missing explicit keyword (missing_explicit_keyword)	Constructor missing the explicit specifier
Missing virtual inheritance (missing_virtual_inheritance)	A base class is inherited virtually and nonvirtually in the same hierarchy
Object slicing (object_slicing)	Derived class object passed by value to function with base class parameter
Partial override of overloaded virtual functions (partial_override)	Class overrides fraction of inherited virtual functions with a given name
Return of non const handle to encapsulated data member (breaking_data_encapsulation)	Method returns pointer or reference to internal member of object
Self assignment not tested in operator (missing_self_assign_test)	Copy assignment operator does not test for self-assignment

Security

Name	Description
Function pointer assigned with absolute address (func_ptr_absolute_addr)	Constant expression is used as function address is vulnerable to code injection

Good Practice

Name	Description
Bitwise and arithmetic operation on the same data (bitwise_arith_mix)	Statement with mixed bitwise and arithmetic operations
Delete of void pointer (delete_of_void_ptr)	delete operates on a void* pointer pointing to an object
Hard-coded buffer size (hard_coded_buffer_size)	Size of memory buffer is a numerical value instead of symbolic constant
Hard-coded loop boundary (hard_coded_loop_boundary)	Loop boundary is a numerical value instead of symbolic constant
Large pass-by-value argument (pass_by_value)	Large argument passed by value between functions
Line with more than one statement (more_than_one_statement)	Multiple statements on a line
Missing break of switch case (missing_switch_break)	No comments at the end of switch case without a break statement
Missing reset of a freed pointer (missing_freed_ptr_reset)	Pointer free not followed by a reset statement to clear leftover data
Unused parameter (unused_parameter)	Function prototype has parameters not read or written in function body

MISRA C:2004 and MISRA AC AGC Rules

The software checks the following rules early in the analysis.

Language Extensions

Rule	Description
2.1	Assembly language shall be encapsulated and isolated.
2.2	Source code shall only use /* */ style comments.
2.3	The character sequence /* shall not be used within a comment.

Documentation

Rule	Description
3.4	All uses of the #pragma directive shall be documented and explained.

Character Sets

Rule	Description
4.1	Only those escape sequences which are defined in the ISO C standard shall be used.
4.2	Trigraphs shall not be used.

Identifiers

Rule	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

Types

Rule	Description
6.1	The plain char type shall be used only for the storage and use of character values.
6.2	Signed and unsigned char type shall be used only for the storage and use of numeric values.
6.3	typedefs that indicate size and signedness should be used in place of the basic types.
6.4	Bit fields shall only be defined to be of type <code>unsigned int</code> or <code>signed int</code> .
6.5	Bit fields of type <code>signed int</code> shall be at least 2 bits long.

Constants

Rule	Description
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and Definitions

Rule	Description
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated.
8.3	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.
8.5	There shall be no definitions of objects or functions in a header file.
8.6	Functions shall always be declared at file scope.
8.7	Objects shall be defined at block scope if they are only accessed from within a single function.
8.8	An external object or function shall be declared in one file and only one file.
8.9	An identifier with external linkage shall have exactly one external definition.
8.11	The <code>static</code> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

Initialization

Rule	Description
9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.
9.3	In an enumerator list, the <code>=</code> construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Arithmetic Type Conversion

Rule	Description
10.1	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: <ul style="list-style-type: none"> • It is not a conversion to a wider integer type of the same signedness, or • The expression is complex, or • The expression is not constant and is a function argument, or • The expression is not constant and is a return expression
10.2	The value of an expression of floating type shall not be implicitly converted to a different type if <ul style="list-style-type: none"> • It is not a conversion to a wider floating type, or • The expression is complex, or • The expression is a function argument, or • The expression is a return expression
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.
10.4	The value of a complex expression of float type may only be cast to narrower floating type.
10.5	If the bitwise operator <code>~</code> and <code><<</code> are applied to an operand of underlying type <code>unsigned char</code> or <code>unsigned short</code> , the result shall be immediately cast to the underlying type of the operand
10.6	The "U" suffix shall be applied to all constants of unsigned types.

Pointer Type Conversion

Rule	Description
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to <code>void</code> .
11.3	A cast should not be performed between a pointer type and an integral type.
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.
11.5	A cast shall not be performed that removes any <code>const</code> or <code>volatile</code> qualification from the type addressed by a pointer

Expressions

Rule	Description
12.1	Limited dependence should be placed on C's operator precedence rules in expressions.
12.3	The <code>sizeof</code> operator should not be used on expressions that contain side effects.
12.5	The operands of a logical <code>&&</code> or <code> </code> shall be primary-expressions.
12.6	Operands of logical operators (<code>&&</code> , <code> </code> and <code>!</code>) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (<code>&&</code> , <code> </code> or <code>!</code>).
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed.
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
12.10	The comma operator shall not be used.
12.11	Evaluation of constant unsigned expression should not lead to wraparound.
12.12	The underlying bit representations of floating-point values shall not be used.
12.13	The increment (<code>++</code>) and decrement (<code>--</code>) operators should not be mixed with other operators in an expression

Control Statement Expressions

Rule	Description
13.1	Assignment operators shall not be used in expressions that yield Boolean values.
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a <code>for</code> statement shall not contain any objects of floating type.
13.5	The three expressions of a <code>for</code> statement shall be concerned only with loop control.
13.6	Numeric variables being used within a <code>for</code> loop for iteration counting should not be modified in the body of the loop.

Control Flow

Rule	Description
14.3	All non-null statements shall either <ul style="list-style-type: none"> • have at least one side effect however executed, or • cause control flow to change.
14.4	The <code>goto</code> statement shall not be used.
14.5	The <code>continue</code> statement shall not be used.
14.6	For any iteration statement, there shall be at most one <code>break</code> statement used for loop termination.
14.7	A function shall have a single point of exit at the end of the function.
14.8	The statement forming the body of a <code>switch</code> , <code>while</code> , <code>do while</code> or <code>for</code> statement shall be a compound statement.
14.9	An <code>if</code> (expression) construct shall be followed by a compound statement. The <code>else</code> keyword shall be followed by either a compound statement, or another <code>if</code> statement.
14.10	All <code>if else if</code> constructs should contain a final <code>else</code> clause.

Switch Statements

Rule	Description
15.0	Unreachable code is detected between <code>switch</code> statement and first <code>case</code> .
15.1	A <code>switch</code> label shall only be used when the most closely-enclosing compound statement is the body of a <code>switch</code> statement
15.2	An unconditional <code>break</code> statement shall terminate every non-empty <code>switch</code> clause.
15.3	The final clause of a <code>switch</code> statement shall be the <code>default</code> clause.
15.4	A <code>switch</code> expression should not represent a value that is effectively Boolean.
15.5	Every <code>switch</code> statement shall have at least one <code>case</code> clause.

Functions

Rule	Description
16.1	Functions shall not be defined with variable numbers of arguments.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.
16.5	Functions with no parameters shall be declared with parameter type <code>void</code> .
16.6	The number of arguments passed to a function shall match the number of parameters.
16.8	All exit paths from a function with non- <code>void</code> return type shall have an explicit return statement with an expression.
16.9	A function identifier shall only be used with either a preceding <code>&</code> , or with a parenthesized parameter list, which may be empty.

Pointers and Arrays

Rule	Description
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	A type should not contain more than 2 levels of pointer indirection.

Structures and Unions

Rule	Description
18.1	All structure or union types shall be complete at the end of a translation unit.
18.4	Unions shall not be used.

Preprocessing Directives

Rule	Description
19.1	<code>#include</code> statements in a file shall only be preceded by other preprocessors directives or comments.
19.2	Nonstandard characters should not occur in header file names in <code>#include</code> directives.
19.3	The <code>#include</code> directive shall be followed by either a <code><filename></code> or <code>"filename"</code> sequence.
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.
19.5	Macros shall not be <code>#define-d</code> and <code>#undef-d</code> within a block.
19.6	<code>#undef</code> shall not be used.
19.7	A function should be used in preference to a function like-macro.
19.8	A function-like macro shall not be invoked without all of its arguments.
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
19.10	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of <code>#</code> or <code>##</code> .
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in <code>#ifdef</code> and <code>#ifndef</code> preprocessor directives and the <code>defined()</code> operator.
19.12	There shall be at most one occurrence of the <code>#</code> or <code>##</code> preprocessor operators in a single macro definition.
19.13	The <code>#</code> and <code>##</code> preprocessor operators should not be used.
19.14	The <code>defined</code> preprocessor operator shall only be used in one of the two standard forms.
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.
19.17	All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> or <code>#ifdef</code> directive to which they are related.

Standard Libraries

Rule	Description
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.
20.2	The names of standard library macros, objects and functions shall not be reused.
20.4	Dynamic heap memory allocation shall not be used.
20.5	The error indicator <code>errno</code> shall not be used.
20.6	The macro <code>offsetof</code> , in library <code><stddef.h></code> , shall not be used.
20.7	The <code>setjmp</code> macro and the <code>longjmp</code> function shall not be used.
20.8	The signal handling facilities of <code><signal.h></code> shall not be used.
20.9	The input/output library <code><stdio.h></code> shall not be used in production code.
20.10	The library functions <code>atof</code> , <code>atoi</code> and <code>atoll</code> from library <code><stdlib.h></code> shall not be used.
20.11	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> from library <code><stdlib.h></code> shall not be used.
20.12	The time handling functions of library <code><time.h></code> shall not be used.

MISRA C:2012 Rules**Standard C Environment**

Rule	Description
1.1	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.
1.2	Language extensions should not be used.

Unused Code

Rule	Description
2.6	A function should not contain unused label declarations.
2.7	There should be no unused parameters in functions.

Comments

Rule	Description
3.1	The character sequences <code>/*</code> and <code>//</code> shall not be used within a comment.
3.2	Line-splicing shall not be used in <code>//</code> comments.

Character Sets and Lexical Conventions

Rule	Description
4.1	Octal and hexadecimal escape sequences shall be terminated.
4.2	Trigraphs should not be used.

Identifiers

Rule	Description
5.2	Identifiers declared in the same scope and name space shall be distinct.
5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
5.4	Macro identifiers shall be distinct.
5.5	Identifiers shall be distinct from macro names.

Types

Rule	Description
6.1	Bit-fields shall only be declared with an appropriate type.
6.2	Single-bit named bit fields shall not be of a signed type.

Literals and Constants

Rule	Description
7.1	Octal constants shall not be used.
7.2	A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type.
7.3	The lowercase character "l" shall not be used in a literal suffix.
7.4	A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

Declarations and Definitions

Rule	Description
8.1	Types shall be explicitly specified.
8.2	Function types shall be in prototype form with named parameters.
8.4	A compatible declaration shall be visible when an object or function with external linkage is defined.
8.5	An external object or function shall be declared once in one and only one file.
8.6	An identifier with external linkage shall have exactly one external definition.
8.8	The <code>static</code> storage class specifier shall be used in all declarations of objects and functions that have internal linkage.
8.10	An inline function shall be declared with the <code>static</code> storage class.
8.11	When an array with external linkage is declared, its size should be explicitly specified.
8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.
8.14	The <code>restrict</code> type qualifier shall not be used.

Initialization

Rule	Description
9.2	The initializer for an aggregate or union shall be enclosed in braces.
9.3	Arrays shall not be partially initialized.
9.4	An element of an object shall not be initialized more than once.
9.5	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

The Essential Type Model

Rule	Description
10.1	Operands shall not be of an inappropriate essential type.
10.2	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.
10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.
10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.
10.5	The value of an expression should not be cast to an inappropriate essential type.
10.6	The value of a composite expression shall not be assigned to an object with wider essential type.
10.7	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.
10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

Pointer Type Conversion

Rule	Description
11.1	Conversions shall not be performed between a pointer to a function and any other type.
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type.
11.3	A cast shall not be performed between a pointer to object type and a pointer to a different object type.
11.4	A conversion should not be performed between a pointer to object and an integer type.
11.5	A conversion should not be performed from pointer to void into pointer to object.
11.6	A cast shall not be performed between pointer to void and an arithmetic type.
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type.
11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.
11.9	The macro NULL shall be the only permitted form of integer null pointer constant.

Expressions

Rule	Description
12.1	The precedence of operators within expressions should be made explicit.
12.3	The comma operator should not be used.
12.4	Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Side Effects

Rule	Description
13.3	A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.
13.4	The result of an assignment operator should not be used.
13.6	The operand of the sizeof operator shall not contain any expression which has potential side effects.

Control Statement Expressions

Rule	Description
14.4	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

Control Flow

Rule	Description
15.1	The goto statement should not be used.
15.2	The goto statement shall jump to a label declared later in the same function.
15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.
15.4	There should be no more than one break or goto statement used to terminate any iteration statement.
15.5	A function should have a single point of exit at the end
15.6	The body of an iteration-statement or a selection-statement shall be a compound statement.
15.7	All if ... else if constructs shall be terminated with an else statement.

Switch Statements

Rule	Description
16.1	All <code>switch</code> statements shall be well-formed.
16.2	A <code>switch</code> label shall only be used when the most closely-enclosing compound statement is the body of a <code>switch</code> statement.
16.3	An unconditional <code>break</code> statement shall terminate every <code>switch</code> -clause.
16.4	Every <code>switch</code> statement shall have a <code>default</code> label.
16.5	A <code>default</code> label shall appear as either the first or the last <code>switch</code> label of a <code>switch</code> statement.
16.6	Every <code>switch</code> statement shall have at least two <code>switch</code> -clauses.
16.7	A <code>switch</code> -expression shall not have essentially Boolean type.

Functions

Rule	Description
17.1	The features of <code><starg.h></code> shall not be used.
17.3	A function shall not be declared implicitly.
17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
17.6	The declaration of an array parameter shall not contain the <code>static</code> keyword between the <code>[]</code> .
17.7	The value returned by a function having non-void return type shall be used.

Pointers and Arrays

Rule	Description
18.4	The <code>+</code> , <code>-</code> , <code>+=</code> and <code>-=</code> operators should not be applied to an expression of pointer type.
18.5	Declarations should contain no more than two levels of pointer nesting.
18.7	Flexible array members shall not be declared.
18.8	Variable-length array types shall not be used.

Overlapping Storage

Rule	Description
19.2	The <code>union</code> keyword should not be used.

Preprocessing Directives

Rule	Description
20.1	<code>#include</code> directives should only be preceded by preprocessor directives or comments.
20.2	The <code>'</code> , <code>"</code> , or <code>\</code> characters and the <code>/*</code> or <code>//</code> character sequences shall not occur in a header file name.
20.3	The <code>#include</code> directive shall be followed by either a <code><filename></code> or <code>"filename\"</code> sequence.
20.4	A macro shall not be defined with the same name as a keyword.
20.5	<code>#undef</code> should not be used.
20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument.
20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.
20.8	The controlling expression of a <code>#if</code> or <code>#elif</code> preprocessing directive shall evaluate to 0 or 1.
20.9	All identifiers used in the controlling expression of <code>#if</code> or <code>#elif</code> preprocessing directives shall be <code>#define</code> 'd before evaluation.
20.10	The <code>#</code> and <code>##</code> preprocessor operators should not be used.
20.11	A macro parameter immediately following a <code>#</code> operator shall not immediately be followed by a <code>##</code> operator.
20.12	A macro parameter used as an operand to the <code>#</code> or <code>##</code> operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.
20.13	A line whose first token is <code>#</code> shall be a valid preprocessing directive.
20.14	All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> , <code>#ifdef</code> or <code>#ifndef</code> directive to which they are related.

Standard Libraries

Rule	Description
21.1	<code>#define</code> and <code>#undef</code> shall not be used on a reserved identifier or reserved macro name.
21.2	A reserved identifier or macro name shall not be declared.
21.3	The memory allocation and deallocation functions of <code><stdlib.h></code> shall not be used.
21.4	The standard header file <code><setjmp.h></code> shall not be used.
21.5	The standard header file <code><signal.h></code> shall not be used.
21.6	The Standard Library input/output functions shall not be used.
21.7	The <code>atof</code> , <code>atoi</code> , <code>atol</code> , and <code>atoll</code> functions of <code><stdlib.h></code> shall not be used.
21.8	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> of <code><stdlib.h></code> shall not be used.
21.9	The library functions <code>bsearch</code> and <code>qsort</code> of <code><stdlib.h></code> shall not be used.
21.10	The Standard Library time and date functions shall not be used.
21.11	The standard header file <code><tgmath.h></code> shall not be used.
21.12	The exception handling features of <code><fenv.h></code> should not be used.

MISRA C++ 2008 Rules

Language Independent Issues

Rule	Description
0-1-7	The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.
0-1-11	There shall be no unused parameters (named or unnamed) in non- virtual functions.
0-1-12	There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.
0-2-1	An object shall not be assigned to an overlapping object.

General

Rule	Description
1-0-1	All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".

Lexical Conventions

Rule	Description
2-3-1	Trigraphs shall not be used.
2-5-1	Digraphs should not be used.
2-7-1	The character sequence /* shall not be used within a C-style comment.
2-10-1	Different identifiers shall be typographically unambiguous.
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
2-10-3	A typedef name (including qualification, if any) shall be a unique identifier.
2-10-4	A class, union or enum name (including qualification, if any) shall be a unique identifier.
2-10-6	If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.
2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.
2-13-2	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.
2-13-3	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.
2-13-4	Literal suffixes shall be upper case.
2-13-5	Narrow and wide string literals shall not be concatenated.

Basic Concepts

Rule	Description
3-1-1	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.
3-1-2	Functions shall not be declared at block scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-1	Objects or functions with external linkage shall be declared in a header file.
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.
3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.
3-9-1	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.
3-9-2	Typedefs that indicate size and signedness should be used in place of the basic numerical types.
3-9-3	The underlying bit representations of floating-point values shall not be used.

Standard Conversions

Rule	Description
4-5-1	Expressions with type <code>bool</code> shall not be used as operands to built-in operators other than the assignment operator <code>=</code> , the logical operators <code>&&</code> , <code> </code> , <code>!</code> , the equality operators <code>==</code> and <code>!=</code> , the unary <code>&</code> operator, and the conditional operator.
4-5-2	Expressions with type <code>enum</code> shall not be used as operands to built-in operators other than the subscript operator <code>[]</code> , the assignment operator <code>=</code> , the equality operators <code>==</code> and <code>!=</code> , the unary <code>&</code> operator, and the relational operators <code><</code> , <code><=</code> , <code>></code> , <code>>=</code> .
4-5-3	Expressions with type (plain) <code>char</code> and <code>wchar_t</code> shall not be used as operands to built-in operators other than the assignment operator <code>=</code> , the equality operators <code>==</code> and <code>!=</code> , and the unary <code>&</code> operator.

Expressions

Rule	Description
5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.
5-0-2	Limited dependence should be placed on C++ operator precedence rules in expressions.
5-0-3	A cvalue expression shall not be implicitly converted to a different underlying type.
5-0-4	An implicit integral conversion shall not change the signedness of the underlying type.
5-0-5	There shall be no implicit floating-integral conversions.
5-0-6	An implicit integral or floating-point conversion shall not reduce the size of the underlying type.
5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression.
5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.
5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.
5-0-10	If the bitwise operators <code>~</code> and <code><<</code> are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.
5-0-11	The plain char type shall only be used for the storage and use of character values.
5-0-12	signed char and unsigned char type shall only be used for the storage and use of numeric values.
5-0-13	The condition of an if-statement and the condition of an iteration-statement shall have type bool.
5-0-14	The first operand of a conditional-operator shall have type bool.
5-0-15	Array indexing shall be the only form of pointer arithmetic.
5-0-18	<code>></code> , <code>>=</code> , <code><</code> , <code><=</code> shall not be applied to objects of pointer type, except where they point to the same array.
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.
5-0-20	Non-constant operands to a binary bitwise operator shall have the same underlying type.
5-0-21	Bitwise operators shall only be applied to operands of unsigned underlying type.
5-2-1	Each operand of a logical <code>&&</code> or <code> </code> shall be a postfix - expression.
5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of <code>dynamic_cast</code> .
5-2-3	Casts from a base class to a derived class should not be performed on polymorphic types.
5-2-4	C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.
5-2-5	A cast shall not remove any const or volatile qualification from the type of a pointer or reference.
5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.
5-2-7	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.

Rule	Description
5-2-10	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.
5-2-11	The comma operator, && operator and the operator shall not be overloaded.
5-2-12	An identifier with array type passed as a function argument shall not decay to a pointer.
5-3-1	Each operand of the ! operator, the logical && or the logical operators shall have type bool.
5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
5-3-3	The unary & operator shall not be overloaded.
5-3-4	Evaluation of the operand to the sizeof operator shall not contain side effects.
5-8-1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.
5-14-1	The right hand operand of a logical && or operator shall not contain side effects.
5-18-1	The comma operator shall not be used.
5-19-1	Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Statements

Rule	Description
6-2-1	Assignment operators shall not be used in sub-expressions.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-2-3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white - space character.
6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.
6-4-1	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.
6-4-2	All if ... else if constructs shall be terminated with an else clause.
6-4-3	A switch statement shall be a well-formed switch statement.
6-4-4	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.
6-4-5	An unconditional throw or break statement shall terminate every non - empty switch-clause.
6-4-6	The final clause of a switch statement shall be the default-clause.
6-4-7	The condition of a switch statement shall not have bool type.
6-4-8	Every switch statement shall have at least one case-clause.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.
6-5-5	A loop-control-variable other than the loop-counter shall not be modified within condition or expression.
6-5-6	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-3	The continue statement shall only be used within a well-formed for loop.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.
6-6-5	A function shall have a single point of exit at the end of the function.

Declarations

Rule	Description
7-3-1	The global namespace shall only contain main, namespace declarations and extern "C" declarations.
7-3-2	The identifier main shall not be used for a function other than the global function main.
7-3-3	There shall be no unnamed namespaces in header files.
7-3-4	using-directives shall not be used.
7-3-5	Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.
7-3-6	using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.
7-4-2	Assembler instructions shall only be introduced using the asm declaration.
7-4-3	Assembly language shall be encapsulated and isolated.

Declarators

Rule	Description
8-0-1	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.
8-3-1	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.
8-4-1	Functions shall not be defined using the ellipsis notation.
8-4-2	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.
8-4-3	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &.
8-5-2	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.
8-5-3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Classes

Rule	Description
9-3-1	const member functions shall not return non-const pointers or references to class-data.
9-3-2	Member functions shall not return non-const handles to class-data.
9-5-1	Unions shall not be used.
9-6-2	Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.
9-6-3	Bit-fields shall not have enum type.
9-6-4	Named bit-fields with signed integer type shall have a length of more than one bit.

Derived Classes

Rule	Description
10-1-1	Classes should not be derived from virtual bases.
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.
10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy.
10-2-1	All accessible entity names within a multiple inheritance hierarchy should be unique.
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

Member Access Control

Rule	Description
11-0-1	Member data in non- POD class types shall be private.

Special Member Functions

Rule	Description
12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor.
12-1-2	All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.
12-1-3	All constructors that are callable with a single argument of fundamental type shall be declared explicit.
12-8-1	A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member.
12-8-2	The copy assignment operator shall be declared protected or private in an abstract class.

Templates

Rule	Description
14-5-2	A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.
14-5-3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.
14-6-1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->.
14-6-2	The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.
14-7-3	All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.
14-8-1	Overloaded function templates shall not be explicitly specialized.
14-8-2	The viable function set for a function call should either contain no function specializations, or only contain function specializations.

Exception Handling

Rule	Description
15-0-2	An exception object should not have pointer type.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-2	NULL shall not be thrown explicitly.
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.
15-3-2	There should be at least one exception handler to catch all otherwise unhandled exceptions
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).

Preprocessing Directives

Rule	Description
16-0-1	#include directives in a file shall only be preceded by other preprocessor directives or comments.
16-0-2	Macros shall only be #define 'd or #undef 'd in the global namespace.
16-0-3	#undef shall not be used.
16-0-4	Function-like macros shall not be defined.
16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.
16-0-7	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.
16-0-8	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.
16-1-1	The defined preprocessor operator shall only be used in one of the two standard forms.
16-1-2	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.
16-2-1	The pre-processor shall only be used for file inclusion and include guards.
16-2-2	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.
16-2-3	Include guards shall be provided.
16-2-4	The ', ", /* or // characters shall not occur in a header file name.
16-2-5	The \ character should not occur in a header file name.
16-2-6	The #include directive shall be followed by either a <filename> or "filename" sequence.
16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition.
16-3-2	The # and ## operators should not be used.
16-6-1	All uses of the #pragma directive shall be documented.
17-0-1	Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.
17-0-2	The names of standard library macros and objects shall not be reused.
17-0-5	The setjmp macro and the longjmp function shall not be used.

Language Support Library

Rule	Description
18-0-1	The C library shall not be used.
18-0-2	The library functions atof, atoi and atol from library <cstdlib> shall not be used.
18-0-3	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.
18-0-4	The time handling functions of library <ctime> shall not be used.
18-0-5	The unbounded functions of library <cstring> shall not be used.
18-2-1	The macro offsetof shall not be used.
18-4-1	Dynamic heap memory allocation shall not be used.
18-7-1	The signal handling facilities of <csignal> shall not be used.

Diagnostic Library

Rule	Description
19-3-1	The error indicator errno shall not be used.

Input/Output Library

Rule	Description
27-0-1	The stream input/output library <cstdio> shall not be used.

Extend CWE Coding Standard Coverage Using Polyspace Defect Checkers

Common Weakness Enumeration (CWE) is a dictionary of common software weakness types that can occur in software architecture, design, code, or implementation. These weaknesses can lead to security vulnerabilities.

Polyspace can check for a subset of CWE rule violations in your code when you enable the option `Check CWE (-cwe)`. In addition to this subset, you can check for additional 95 CWE rule violations by enabling Polyspace defect checkers that you map to these additional CWE rules.

f

Find CWE IDs from Polyspace Results

Use this workflow to check for CWE violations explicitly and extend the CWE coverage using mapped results.

- *Analysis:*

Enable a subset of CWE rules using option `Check CWE (-cwe)` with parameter `all`, `cwe-658-659`, or `custom`. See “Common Weakness Enumeration (CWE)”.

Use option `Find defects (-checkers)` with parameter `custom` to enable the defect checkers that correspond to the additional CWE rules that you want to check.

- *Results:*

The CWE violations that correspond to the rules that you enabled with option `-cwe` appear as coding rule violations in the **Results List**.

In the Defects family of results you see only the defects that correspond to mapped CWE rules.

Tip If you checked your code for CWE violations using a Polyspace product version R2022b or earlier, the CWE ID column in the **Results List** is no longer available when your review results generated with a Polyspace product version R2023a. You can track which CWE rule corresponds to which checker in your results by, for instance, adding the CWE ID of the rule as a comment for the corresponding defect.

Mapping Between CWE Identifiers and Polyspace Results

This table lists the CWE IDs (version 4.9) that you can map to Polyspace defect checkers. You can explicitly check for 101 CWE rule violations and an additional 95 CWE rule violations using Polyspace Bug Finder defect checkers.

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
15	External control of system or configuration setting	Host change using externally controlled elements Use of externally controlled environment variable
20	Improper input validation	Unsafe conversion from string to numerical value
22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	Vulnerable path manipulation
23	Relative path traversal	Vulnerable path manipulation
36	Absolute path traversal	Vulnerable path manipulation
67	Improper Handling of Windows Device Names	Inappropriate I/O operation on device files
77	Improper neutralization of special elements used in a command	Execution of externally controlled command
78	Improper neutralization of special elements used in an OS command	Execution of externally controlled command Unsafe call to a system function
88	Argument injection or modification	Execution of externally controlled command Unsafe call to a system function
114	Process control	Command executed from externally controlled path Execution of a binary from a relative path can be controlled by an external actor Execution of externally controlled command Library loaded from externally controlled path Load of library from a relative path can be controlled by an external actor

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
189	Numeric Errors	Absorption of float operand Float conversion overflow Float division by zero Float overflow Integer constant overflow Integer conversion overflow Integer division by zero Integer overflow Precision loss in integer to float conversion Right operand of shift operation outside allowed bounds Shift of a negative value Tainted division operand Unsigned integer constant overflow Unsigned integer conversion overflow Unsigned integer overflow
190	Integer overflow or wraparound	Integer constant overflow Integer conversion overflow Integer overflow Integer precision exceeded Possible invalid operation on boolean operand Right operand of shift operation outside allowed bounds Tainted division operand Unsigned integer constant overflow Unsigned integer conversion overflow Unsigned integer overflow
198		Missing byte reordering when transferring data

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
226	Sensitive information uncleared before release	Uncleared sensitive data in stack
227	Improper fulfillment of API contract	Invalid use of standard library floating point routine Invalid use of standard library integer routine Invalid use of standard library memory routine Invalid use of standard library routine Invalid use of standard library string routine Writing to const qualified object
240	Improper handling of inconsistent structural elements	Mismatch between data length and size
250	Execution with unnecessary privileges	Bad order of dropping privileges Privilege drop not verified
251	Often misused: string management	Destination buffer overflow in string manipulation
273	Improper check for dropped privileges	Privilege drop not verified
287	Improper Authentication	X.509 peer certificate not checked
297	Improper Validation of Certificate with Host Mismatch	Server certificate common name not checked
304	Missing Critical Step in Authentication	TLS/SSL connection method not set

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
310	Cryptographic issues	Constant block cipher initialization vector Constant cipher key Context initialized incorrectly for cryptographic operation Context initialized incorrectly for digest operation Incompatible padding for RSA algorithm operation Incorrect key for cryptographic algorithm Missing blinding for RSA algorithm Missing block cipher initialization vector Missing certification authority list Missing cipher algorithm Missing cipher key Missing data for encryption, decryption or signing operation Missing padding for RSA algorithm Missing parameters for key generation Missing peer key Missing private key Missing public key Missing X.509 certificate Nonsecure hash algorithm Nonsecure parameters for key generation Nonsecure RSA public exponent Nonsecure SSL/TLS protocol Predictable block cipher initialization vector Predictable cipher key Weak cipher algorithm

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
		Weak cipher mode Weak padding for RSA algorithm
311	Missing encryption of sensitive data	Missing cipher data to process Missing cipher final step
316	Cleartext Storage of Sensitive Information in Memory	Sensitive heap memory not cleared before release Uncleared sensitive data in stack
320	Key management errors	Constant cipher key Missing cipher key Missing peer key Missing private key Missing public key
321	Use of hard-coded cryptographic key	Constant cipher key
322	Key Exchange without Entity Authentication	TLS/SSL connection method not set
325	Missing required cryptographic step	Context initialized incorrectly for cryptographic operation Incorrect key for cryptographic algorithm Missing cipher data to process Missing cipher final step Missing data for encryption, decryption or signing operation Missing parameters for key generation No data added into context

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
326	Inadequate encryption strength	Constant block cipher initialization vector Constant cipher key Missing blinding for RSA algorithm Missing block cipher initialization vector Missing padding for RSA algorithm Nonsecure parameters for key generation Nonsecure RSA public exponent Predictable cipher key Weak cipher algorithm Weak cipher mode Weak padding for RSA algorithm
327	Use of a broken or risky cryptographic algorithm	Missing padding for RSA algorithm Nonsecure hash algorithm Nonsecure parameters for key generation Nonsecure RSA public exponent Nonsecure SSL/TLS protocol Unsafe standard encryption function Weak cipher algorithm Weak cipher mode Weak padding for RSA algorithm
328	Reversible one-way hash	Nonsecure hash algorithm
329	Not using a random IV with CBC mode	Constant block cipher initialization vector Missing block cipher initialization vector Predictable block cipher initialization vector

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
330	Use of insufficiently random values	<p>Deterministic random output from constant seed</p> <p>Predictable block cipher initialization vector</p> <p>Predictable cipher key</p> <p>Predictable random output from predictable seed</p> <p>Vulnerable pseudo-random number generator</p>
336	Same seed in PRNG	Deterministic random output from constant seed
337	Predictable seed in PRNG	Predictable random output from predictable seed
354	Improper Validation of Integrity Check Value	Context initialized incorrectly for digest operation
367	Time-of-check time-of-use (TOCTOU) race condition	File access between time of check and use (TOCTOU)
372	Incomplete internal state distinction	<p>Context initialized incorrectly for cryptographic operation</p> <p>Context initialized incorrectly for digest operation</p> <p>Incompatible padding for RSA algorithm operation</p> <p>Inconsistent cipher operations</p> <p>Missing cipher data to process</p> <p>Missing cipher final step</p> <p>Missing data for encryption, decryption or signing operation</p> <p>Missing parameters for key generation</p>
377	Insecure temporary file	Use of non-secure temporary file

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
387	Signal errors	Function called from signal handler not asynchronous-safe Function called from signal handler not asynchronous-safe (strict ISO C) Return from computational exception signal handler Signal call from within signal handler
391	Unchecked error condition	Errno not checked
398	Indicator of poor code quality	Write without a further read
404	Improper resource shutdown or release	Invalid deletion of pointer Invalid free of pointer Memory leak Mismatched alloc/dealloc functions on Windows Thread-specific memory leak
426	Untrusted search path	Command executed from externally controlled path Library loaded from externally controlled path
427	Uncontrolled search path element	Execution of a binary from a relative path can be controlled by an external actor Load of library from a relative path can be controlled by an external actor
456	Missing initialization of a variable	Errno not reset Member not initialized in constructor Non-initialized pointer Non-initialized variable
465	Pointer Issues	Unsafe conversion between pointer and integer
471	Modification of assumed-immutable data	Writing to const qualified object
475	Undefined behavior for input to API	Copy of overlapping memory
532	Information exposure through log files	Sensitive data printed out

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
534	Information exposure through debug log files	Sensitive data printed out
535	Information exposure through shell error message	Sensitive data printed out
573	Improper following of specification by caller	<p>Context initialized incorrectly for cryptographic operation</p> <p>Context initialized incorrectly for digest operation</p> <p>Incompatible padding for RSA algorithm operation</p> <p>Incorrect key for cryptographic algorithm</p> <p>Missing blinding for RSA algorithm</p> <p>Missing cipher algorithm</p> <p>Missing cipher key</p> <p>Missing data for encryption, decryption or signing operation</p> <p>Missing final step after hashing update operation</p> <p>Missing hash algorithm</p> <p>Missing parameters for key generation</p> <p>Missing peer key</p> <p>Missing private key</p> <p>Missing private key for X.509 certificate</p> <p>Missing public key</p> <p>Modification of internal buffer returned from non-reentrant standard function</p> <p>TLS/SSL connection method not set</p> <p>TLS/SSL connection method set incorrectly</p>
590	Free of memory not on the heap	Invalid free of pointer
606	Unchecked input for loop condition	Loop bounded with tainted value

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
628	Function call with incorrectly specified arguments	Bad file access mode or status Copy of overlapping memory Invalid va_list argument Modification of internal buffer returned from non-reentrant standard function Standard function call with incorrect arguments
663	Use of a non-reentrant function in a concurrent context	Function called from signal handler not asynchronous-safe Function called from signal handler not asynchronous-safe (strict ISO C) Unsafe standard encryption function Unsafe standard function
664	Improper control of a resource through its lifetime	Context initialized incorrectly for cryptographic operation Context initialized incorrectly for digest operation Incompatible padding for RSA algorithm operation Inconsistent cipher operations Incorrect key for cryptographic algorithm Missing cipher data to process Missing cipher final step Missing cipher key Missing peer key Missing private key Missing public key
665	Improper initialization	Call to memset family with unintended value Improper array initialization Overlapping assignment Use of memset with size argument zero

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
666	Operation on resource in wrong phase of lifetime	Incorrect order of network connection operations
667	Improper locking	Blocking operation while holding lock Destruction of locked mutex Missing unlock
672	Operation on a resource after expiration or release	Closing previously closed resource Use of previously closed resource
675	Duplicate operations on resource	Opening previously opened resource
681	Incorrect conversion between numeric types	Float conversion overflow Precision loss in integer to float conversion
682	Incorrect calculation	Absorption of float operand Bitwise operation on negative value Float overflow Invalid use of standard library floating point routine Invalid use of standard library integer routine Tainted modulo operand Use of plain char type for numeric value
683	Function Call With Incorrect Order of Arguments	Call to memset family with unintended value Format string specifiers and arguments mismatch

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
686	Function call with incorrect argument type	Bad file access mode or status Declaration mismatch Format string specifiers and arguments mismatch Incorrect data type passed to va_arg Standard function call with incorrect arguments Use of automatic variable as putenv-family function argument Writing to const qualified object
687	Function call with incorrectly specified argument value	Copy of overlapping memory Standard function call with incorrect arguments Variable length array with non-positive size
691	Insufficient control flow management	Use of setjmp/longjmp
693	Protection mechanism failure	Nonsecure SSL/TLS protocol
696	Incorrect behavior order	Bad order of dropping privileges
703	Improper check or handling of exceptional conditions	Errno not reset Misuse of errno
705	Incorrect control flow scoping	Abnormal termination of exit handler
710	Coding standard violation	Bitwise and arithmetic operations on the same data
732	Incorrect permission assignment for critical resource	Vulnerable permission assignments
754	Improper check for unusual or exceptional conditions	Returned value of a sensitive function not checked
755	Improper handling of exceptional conditions	Exception handler hidden by previous handler

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
758	Reliance on undefined, unspecified, or implementation-defined behavior	Bitwise operation on negative value Unsafe conversion between pointer and integer Use of plain char type for numeric value
759	Use of a One-Way Hash without a Salt	Missing salt for hashing operation
764	Multiple locks of a critical resource	Double lock
765	Multiple unlocks of a critical resource	Double unlock
770	Allocation of resources without limits or throttling	Tainted size of variable length array
772	Missing release of resource after effective lifetime	Resource leak
780	Use of rsa algorithm without oaep	Missing padding for RSA algorithm Weak padding for RSA algorithm
786	Access of memory location before start of buffer	Destination buffer underflow in string manipulation
822	Untrusted pointer dereference	Tainted NULL or non-null-terminated string Use of tainted pointer
823	Use of out-of-range pointer offset	Pointer access out of bounds Pointer dereference with tainted offset
826	Premature release of resource during expected lifetime	Closing previously closed resource Destruction of locked mutex Use of previously closed resource
828	Signal handler with functionality that is not asynchronous-safe	Function called from signal handler not asynchronous-safe Function called from signal handler not asynchronous-safe (strict ISO C)
832	Unlock of a resource that is not locked	Missing lock
833	Deadlock	Deadlock
872	CERT C++ Secure Coding Section 04 - Integers (INT)	Invalid use of standard library integer routine

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
873	CERT C++ Secure Coding Section 05 - Floating point arithmetic (FLP)	Absorption of float operand Float overflow Floating point comparison with equality operators Invalid use of standard library floating point routine
908	Use of uninitialized resource	Member not initialized in constructor Non-initialized pointer Non-initialized variable

Configure File Sets for Bug Finder Analysis

Classify Project Files Into File Sets for Precise Control of Bug Finder Analysis

When analyzing C/C++ code with Polyspace Bug Finder, you can define file sets in your project that need specific treatment during analysis. For instance, you might want to skip the definitions of function bodies in third-party libraries or force analysis of all functions in files that you own. You can enumerate file sets with specific behaviors in a classification XML file and fine-tune the Bug Finder analysis using this classification file.

This example shows how to create a classification file and breaks down the structure of such a file. In your projects, you can apply similar reasoning to partition files into file sets and create appropriate classification files.

Classification File Structure Based on Analysis Requirements

Typically, the file sets in your classification file reflect an actual folder structure (or file name patterns).

This example follows the structure shown in the sample classification file in *polyspaceroot* \polyspace\verifier\cxx\classification-template.xml (here, *polyspaceroot* is the Polyspace installation folder, for instance, C:\Program Files\Polyspace\R2023a). The classification file assumes a folder structure with these top-level folders and these source and header files. The nature of these files imposes requirements on how they must be analyzed.

Folder	Description	Files	Analysis Requirements
myproject	Folder containing the primary sources and headers to analyze. Some header files are machine-generated. The names of these files end with the string -generated.	<ul style="list-style-type: none"> • myproject/src/foo.cpp • myproject/src/foo.hpp • myproject/src/main.cpp • myproject/inc/bar-generated.hpp • myproject/inc/bar.hpp 	All files excluding machine-generated ones must be fully analyzed. The analysis results must be available for review.
acmelib	Folder containing headers from a third-party library.	acmelib/inc/acme.hpp	All files must be included in analysis, but results must be suppressed from review. (Even if issues are found in third-party libraries, they cannot be typically fixed.)

Folder	Description	Files	Analysis Requirements
usr	Folder containing system headers.	usr/include/sys/os.h	All files must be excluded from analysis. This means that only function declarations must be used in the analysis but function bodies discarded.

These requirements lead to the following file sets in the classification XML file. Each file set is defined through a `fileset` element inside the classification file:

```
<?xml version="1.0" encoding="UTF-8"?>
<specification>
  <classification product="bug-finder">
    <fileset name="Application implementation and header files">
      <!--File set definitions-->
    </fileset>
    <fileset name="Generated code">
      <!--File set definitions-->
    </fileset>
    <fileset name="Third-party libraries">
      <!--File set definitions-->
    </fileset>
    <fileset name="Everything else">
      <!--File set definitions-->
    </fileset>
  </classification>
</specification>
```

Note that there can be multiple `classification` elements inside the file as long as their `product` attributes are different. The `product` attribute takes two values:

- `bug-finder`: The file sets in this `classification` element applies to a full Polyspace Bug Finder analysis.
- `bug-finder-access`: The file sets in this `classification` element applies to a single-file analysis with Polyspace as You Code.

Classification File Usage

You specify a classification XML file using the Polyspace analysis option `-classification`. If you are running an analysis in the Polyspace user interface (desktop products only), you have to enter this option in the `Other` field.

For instance, suppose that your classification file is named `my-classification-template.xml` and is located in the folder that contains the top-level source folders listed previously. You can specify the classification file in a Polyspace options file on page 12-5 written as follows:

```
-sources myproject/src/foo.cpp
-sources myproject/src/main.cpp
-I myproject/inc
-I acmelib/inc
-I usr/include/sys
-classification my-classification-template.xml
```

This options file is assumed to be in the same folder as `my-classification-template.xml`. The options file specifies:

- Which source files must be analyzed (using option `-sources`).
- Which folders must be used for header file lookup (using option `-I`).
- Which classification XML file must be used to partition source and header files into file sets (using option `-classification`).

If the options file is named `psoptions.txt`, you can provide this options file to a Polyspace Bug Finder analysis as:

```
polyspace-bug-finder -options-file psoptions.txt
```

(An options file - a text file with one analysis option per line - is used in this example for convenience. You can also specify the analysis options directly at the command line with the `polyspace-bug-finder` command.)

Parts of Classification File

The classification file contains several file sets, defined through `fileset` elements. Each file set defines:

- A set of files to include in a `files-in-set` element.

For more information on how to specify files, see “Select Files for Polyspace Analysis Using Pattern Matching” on page 4-11.

- A set of files to exclude in a `files-not-in-set` element.

For more information on how to specify files, see “Select Files for Polyspace Analysis Using Pattern Matching” on page 4-11.

- A set of behaviors to associate in a `behaviors` element. The currently supported behaviors are defined through these elements:

- `analyze-functions` and `do-not-analyze-functions`: Using this element, you specify patterns for functions that must and must not be analyzed. You specify the patterns in a `function-pattern` element. The patterns currently supported are:

Function Pattern	Meaning
*	All functions
:: <i>top_level_namespace</i> ::	All functions in top level namespace <i>top_level_namespace</i>
\$(all-relevant-functions)	All non-static functions and all static functions called somewhere in the code, excluding some pathological cases.

- `show-results`: Using this element, you specify whether you want to see results in the files that are included in the file set. If if you do not want to see results, you set the `value` attribute of this element to `false`.

Note that the `show-results` element applies to a result only if all files involved in the result are part of the file set. If a result involves two files, one in the file set and another not in the file set, and you specify a value of `false` for this file set, you might still see the result.

File Set 1: Application Sources and Headers

The first file set in your classification file typically defines the primary set of sources and headers to analyze. In this example, the `fileset` element contains the following file set definitions:

```
<fileset name="Application implementation and header files">
  <files-in-set>
    <file-pattern>$(source-files)</file-pattern>
    <file-pattern>$(source-headers)</file-pattern>
    <file-pattern>myproject/inc/**/* .hpp</file-pattern>
  </files-in-set>
  <files-not-in-set>
    <file-pattern>myproject/**/*-generated.hpp</file-pattern>
  </files-not-in-set>
  <behaviors>
    <analyze-functions>
      <function-pattern>*</function-pattern>
    </analyze-functions>
    <do-not-analyze-functions>
      <function-pattern>::self_test::</function-pattern>
    </do-not-analyze-functions>
    <show-results value="true"/>
  </behaviors>
</fileset>
```

Files Included

This file set includes the following files in separate `file-pattern` elements:

- All source files specified using the option `-sources`. The inclusion is done using the pattern `$(source-files)`.
- Header files in the same folders as source files or in subfolders thereof. The inclusion is done using the pattern `$(source-headers)`.
- Header files in the folder `myproject/inc` or in subfolders thereof. The inclusion is done using the pattern `myproject/inc/**/* .hpp`. The `**` indicates recursive file lookup in subfolders.

Files Excluded

This file set excludes all `.hpp` files in the folder `myproject` or in subfolders thereof, with names ending in the string `-generated`. The exclusion is done using the pattern `myproject/**/*-generated.hpp`. The `**` indicates recursive file lookup in subfolders.

Behaviors

This file set has the following behaviors:

- All functions in the included files except ones in the C++ namespace `self_test` are analyzed.

A function is analyzed if it matches a pattern in the `analyze-functions` element and does not match any pattern in the `do-not-analyze-functions` element. The pattern `*` indicates *all* functions, even ones that Bug Finder would normally skip, for instance, uncalled static functions.
- All results in the included files are shown.

File Set 2: Generated Files

The second file set in this example covers generated files. The `fileset` element contains the following file set definitions:

```

<fileset name="Generated code">
  <files-in-set>
    <file-pattern>myproject/**/*-generated.hpp</file-pattern>
  </files-in-set>
  <behaviors>
    <analyze-functions>
      <function-pattern>$(all-relevant-functions)</function-pattern>
    </analyze-functions>
    <show-results value="false"/>
  </behaviors>
</fileset>

```

Files Included

This file set includes all .hpp files in the folder `myproject` or in subfolders thereof, with names ending in the string `-generated`. The inclusion is done using the pattern `myproject/**/*-generated.hpp`. The `**` indicates recursive file lookup in subfolders.

Behaviors

This file set has the following behaviors:

- All functions in the included files are analyzed, except ones that Bug Finder does not analyze by default (static uncalled functions and some pathological functions, for instance, ones containing several thousands of instructions).

The pattern `$(all-relevant-functions)` indicates this default subset of functions.

- Results in the included files are not shown.

The behaviors follow from the fact that functions in generated code can contribute to issues in handwritten code and must be included in the analysis. However, if the issues occur in the generated code itself, they are typically difficult to fix.

File Set 3: Third-Party Libraries

The third file set in this example covers third-party libraries. The `fileset` element contains the following file set definitions:

```

<fileset name="Third-party libraries">
  <files-in-set>
    <file-pattern>acmelib/**</file-pattern>
  </files-in-set>
  <behaviors>
    <analyze-functions>
      <function-pattern>$(all-relevant-functions)</function-pattern>
    </analyze-functions>
    <show-results value="false"/>
  </behaviors>
</fileset>

```

Files Included

This file set includes all files in the folder `acmelib` or in subfolders thereof. The inclusion is done using the pattern `acmelib/**/`. The `**` indicates recursive file lookup in subfolders.

Behaviors

This file set has the following behaviors:

- All functions in the included files are analyzed, except ones that Bug Finder does not analyze by default (static uncalled functions and some pathological functions, for instance, ones containing several thousands of instructions).
- Results in the included files are not shown.

The behaviors follow from the fact that functions in third-party libraries can contribute to issues in the application code and must be included in the analysis. However, if the issues occur in the third-party libraries themselves, they are typically difficult to fix. This file set has the same behaviors as the generated code file set. However, a separate file set has been created to explicitly separate file patterns in generated code from ones in third-party libraries. This separation can make later maintenance easier.

File Set 4: Everything Else (Typically System Headers)

The fourth file set in this example is a catch-all subset that covers files not covered by included patterns in previous file sets. The `fileset` element contains the following file set definitions:

```
<fileset name="Everything else">
  <files-in-set>
    <file-pattern>*</file-pattern>
  </files-in-set>
  <behaviors>
    <do-not-analyze-functions>
      <function-pattern>*</function-pattern>
    </do-not-analyze-functions>
    <show-results value="false"/>
  </behaviors>
</fileset>
```

Files Included

This file set includes all files not included by any of the previous file sets. (Note that the analysis tries to match file sets in order, so if a file matched a pattern in a previous file set, it will not fall through to this file set.)

This file set typically includes system headers. In this example, the system headers are in the folder `usr/include/sys` and are matched by the catch-all file pattern `*`.

Behaviors

This file set has the following behaviors:

- The bodies of functions in the included files are not analyzed.
- Results in the included files are not shown.

The behaviors follow from the fact that system libraries typically contain many inline functions that could slow down the analysis. The behaviors make the analysis only consider function declarations but skip the function bodies.

See Also

-classification

Related Examples

- “Select Files for Polyspace Analysis Using Pattern Matching” on page 4-11


Configure Comment Import from Previous Results

- “Import Review Information from Previous Polyspace Analysis” on page 20-2
- “Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results” on page 20-5

Import Review Information from Previous Polyspace Analysis

This topic describes how to import review information from previous results that are not already uploaded to Polyspace Access. For information on importing from results uploaded to Polyspace Access, see “Import Review Information from Existing Polyspace Access Projects” on page 26-5.

After you have reviewed analysis results, you can reuse information from the review for subsequent analyses. If you specify a result status or severity or add notes in your results file, they carry over to the results of the next analysis on the same project. If you add the same information as comments to your code (annotate), they carry over to any subsequent analysis of the code, whether in the same project or not. You can also hide results using code annotations. For more information on commenting, see “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 22-2.

This topic shows how to import review information from one result file to another. Importing the review information saves you from reviewing already justified results. For instance, after you import the information, on the **Results List** pane (user interface of desktop products), clicking the  icon skips justified results. Using this icon, you can browse through unreviewed results. You can also filter the justified checks from display.

Automatic Import from Last Analysis

By default, in the Polyspace user interface (desktop products only), review information is imported automatically from the most recent analysis on the project module. You can disable this default behavior.

- 1 Select **Tools > Preferences**, which opens the Polyspace Preferences dialog box.
- 2 Select the **Project and Results Folder** tab.
- 3 Under **Import Comments**, clear **Automatically import comments from last verification**.
- 4 Click **OK**.

If you run analysis at the command line (and do not upload results to the Polyspace Access web interface), you have to explicitly import from another set of results. See “Command Line” on page 20-3.

Import from Another Analysis Result

You can import review information directly from another Polyspace result to the current result.

If a result is found in both a Bug Finder and Code Prover analysis, you can add review information to the Bug Finder result and import to the Code Prover result. For instance, most coding rule checkers are common to Bug Finder and Code Prover. You can add review information to coding rule violations in Bug Finder and import to the same violations in Code Prover.

User Interface (Desktop Products Only)

To import review information from another set of results:

- 1 Open the current analysis results.
- 2 Select **Tools > Import Comments**.

- 3 Navigate to the folder containing your previous results.
- 4 Select the other results file (with extension `.psbf` or `.pscp`) and then click **Open**.

The review information from the previous results are imported into the current results.

Command Line

Use the option `-import-comments` during analysis to import comments from a previous verification.

To import review information from multiple results, use the `polyspace-comments-import` command.

Import Algorithm

You can directly import review information from another set of results into the current results. However, it is possible that part of your review information is not imported to a subsequent analysis because:

- You have changed your source code so that the line with a previous result is not exactly identical to the line in the current run.

The comment import tool accounts for additional code that simply shifts an existing line. For instance, the tool recognizes that line 10 in Run 1 and line 12 in Run 2 have the same statement. If a division by zero occurs on line 10 in Run 1 and you have not fixed the issue in Run 2, the result along with associated review information are imported to line 12 in Run 2.

- Run 1:

```
10 baseLine = min/numRecipients;
11
12
```

- Run 2:

```
10 /* Calculate a baseline per recipient
11    based on minimum available resources */
12 baseLine = min/numRecipients;
```

However, if you change the line content itself, for instance, change `numRecipient` to `numReceiver`, the result and review information are not imported.

- You have changed your source code so that the Code Prover result color has changed.
- You entered new review information for the same result.

If the content of a line does not change and shows the same result as the previous analysis, the review information is imported. In unlikely scenarios, you might get the same result on the same line despite changing previous lines that lead to the result. Your review information from a previous analysis is then imported to the new result. If you justified the previous result with a status such as **Not a defect**, it is likely that you want to continue this justification with the new result. For instance, if you accepted an overflow previously because you accounted for a wrap-around behavior after the overflow, you are likely to accept the overflow whatever the cause. In a few cases, you might want to review the result again and might not be aware that the result merits another review. To avoid this situation:

- When justifying nonlocal results that are related to previous events, use careful judgement.

- For critical components, conduct periodic assessments of justified results to see if the justifications still apply. Such assessments are useful specially for the Code Prover run-time checks.

View Imported Review Information That Does Not Apply

In the Polyspace user interface (desktop products only), the Import Checks and Comments Report highlights differences between two analysis results. When you import review information from a previous analysis, you can see this report. If you have closed the report after an import, to review the report again:

- 1 Select **Window > Show/Hide View > Import Comments Report**.

The Import Checks and Comments Report opens, highlighting differences in the two results.

File	Function	Line	Col	Check	Import details	Justified	Classification	Status	Comment
example.c	example.c	11		2/CVFL	Check color has changed from Green to Orange	<input checked="" type="checkbox"/>	Not a defect	No action planned	This might overflow.

- 2 Review the differences between the two results.

Your review information can differ between two results because of the following reasons:

- In Code Prover, if the check color changes, Polyspace imports the **Comment** field but not the **Status** field. In addition, Polyspace imports the **Severity** and **Justified** fields depending on the color change.

Color Change	Severity	Justified
Orange or red to green	Not imported	Imported
Gray to green	Not imported	Imported, if the Severity was set to High, Medium or Low.
Red to orange or vice versa	Imported	Imported
Green to red/orange/gray	Not imported	Not imported

- If a result no longer appears in the code, Polyspace highlights only the change in the Import Checks and Comments Report. It does not import review information from the previous result.
- If you have already entered different review information for the same check, Polyspace highlights only the change in the Import Checks and Comments Report. It does not import review information from the previous result.

See Also

-import-comments | polyspace-comments-import

Related Examples

- “Import Review Information from Existing Polyspace Access Projects” on page 26-5

Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results

When you check your code for MISRA C: 2012 violations, Polyspace imports justifications of MISRA C: 2004 violations from previous analyses (if they exist). You can upgrade from checking of MISRA C: 2004 rules to MISRA C: 2012 rules while retaining your justifications. For general rules on comment import, see “Import Review Information from Previous Polyspace Analysis” on page 20-2.

The software maps MISRA C: 2004 **Status**, **Severity**, and **Comment** values that you added through the user interface or code annotations to the corresponding MISRA C: 2012 results, if the results exist. For more information about mapping, consult addendum one of the MISRA C: 2012 publication.

Type	Check: (9)	Status	Severity	Comment: (9)
MISRA C:2004	6.3 Typedefs that indicate size and sig...	Unreviewed	Unset	MISRA2004-6.3 comment
MISRA C:2004	6.3 Typedefs that indicate size and sig...	To fix	Medium	MISRA2004-6.3
MISRA C:2004	8.1 Functions shall have prototype de...	To fix	Low	MISRA2004-8.1
MISRA C:2004	11.3 A cast should not be performed b...	Justified	Low	MISRA2004-11.3
MISRA C:2004	11.4 A cast should not be performed b...	Unreviewed	Unset	MISRA2004-11.4 comment
MISRA C:2004	12.12 The underlying bit representatio...	Unreviewed	Unset	MISRA2004-12.12 comm...
MISRA C:2004	13.2 Tests of a value against zero sho...	Not a defect	Low	MISRA2004-13.2
MISRA C:2004	14.4 The goto statement shall not be ...	Not a defect	Low	MISRA2004-14.4
MISRA C:2004	14.9 An if (expression) construct shall ...	Not a defect	Low	MISRA2004-13.2
MISRA C:2004	19.5 Macros shall not be #define'd an...	Justified	Low	MISRA2004-19.5

If you are transitioning from MISRA C: 2004 to MISRA C: 2012, you do not have to review results that you have already justified.

Type	Check	Status	Severity	Comment: (7)
MISRA C:2012	Dir 4.6 typedefs that indicate size and...	Unreviewed	Unset	MISRA2004-6.3 comment
MISRA C:2012	Dir 4.6 typedefs that indicate size and...	To fix	Medium	MISRA2004-6.3
MISRA C:2012	8.4 A compatible declaration shall be v...	To fix	Low	MISRA2004-8.1
MISRA C:2012	11.3 A cast shall not be performed bet...	Unreviewed	Unset	MISRA2004-11.4 comment
MISRA C:2012	11.4 A conversion should not be perfo...	Justified	Low	MISRA2004-11.3
MISRA C:2012	14.4 The controlling expression of an i...	Not a defect	Low	MISRA2004-13.2
MISRA C:2012	15.1 The goto statement should not b...	Not a defect	Low	MISRA2004-14.4
MISRA C:2012	15.6 The body of an iteration-stateme...	Not a defect	Low	MISRA2004-13.2

Mapping Multiple MISRA C: 2004 Annotations to the Same MISRA C: 2012 Result

When you justify MISRA C: 2004 violations by using code block syntax or multiple line annotation syntax, and multiple violations map to the same MISRA C: 2012 rule, Polyspace does not import each result justification. Instead, the software imports only one set of **Status**, **Severity**, and **Comment** values and applies these values to all the instances of that particular MISRA C: 2012 rule violation.

For example, suppose that you analyze your code and find violations of MISRA C: 2004 rules 16.3 and 16.5. You can justify these results by using the annotation syntax where you enter a different status and explanatory comment for each rule.

```
//polyspace-begin misra2004:16.3 [Status 1] "Explanatory comment 1"  
//polyspace-begin misra2004:16.5 [Status 2] "Explanatory comment 2"  
  
code block start;  
/* This block of code contains violations of  
MISRA C:2004 rules 16.3 and 16.5 */  
code block end;  
  
//polyspace-end misra2004:16.3  
//polyspace-end misra2004:16.5
```

The previous violations map to MISRA C: 2012 rule 8.2. When you check your annotated code against MISRA C: 2012 rules, Polyspace imports only the first line of annotations (for rule 16.3) and applies it to all rule 8.2 results. The second line of annotations for rule 16.5 is ignored. In the **Results List** pane, all violations of rule 8.2 have the **Status** column set to **Status 1** and the **Comment** column set to **"Explanatory comment 1"**.

Note The **Output Summary** pane displays a warning message for every result where the imported annotation conflicts with the original annotation. After you import your MISRA C: 2004 annotations, check that a justified status has not been assigned to results you intend to investigate or fix.

See Also

Check MISRA C:2004 (-misra2) | Check MISRA C:2012 (-misra3)

More About

- "Annotate Code and Hide Known or Acceptable Results" on page 30-2

Review Results in Polyspace User Interface

Interpret Polyspace Bug Finder Results

- “Interpret Bug Finder Results in Polyspace Desktop User Interface” on page 21-2
- “Investigate the Cause of Empty Results List” on page 21-7
- “Dashboard in Polyspace Desktop User Interface” on page 21-9
- “Concurrency Modeling in Polyspace Desktop User Interface” on page 21-13
- “Results List in Polyspace Desktop User Interface” on page 21-15
- “Source Code in Polyspace Desktop User Interface” on page 21-17
- “Result Details in Polyspace Desktop User Interface” on page 21-22
- “Call Hierarchy in Polyspace Desktop User Interface” on page 21-24
- “Understanding Changes in Polyspace Results After Product Upgrade” on page 21-26


Interpret Bug Finder Results in Polyspace Desktop User Interface

This topic shows how to review Bug Finder results in the user interface of the Polyspace desktop products.

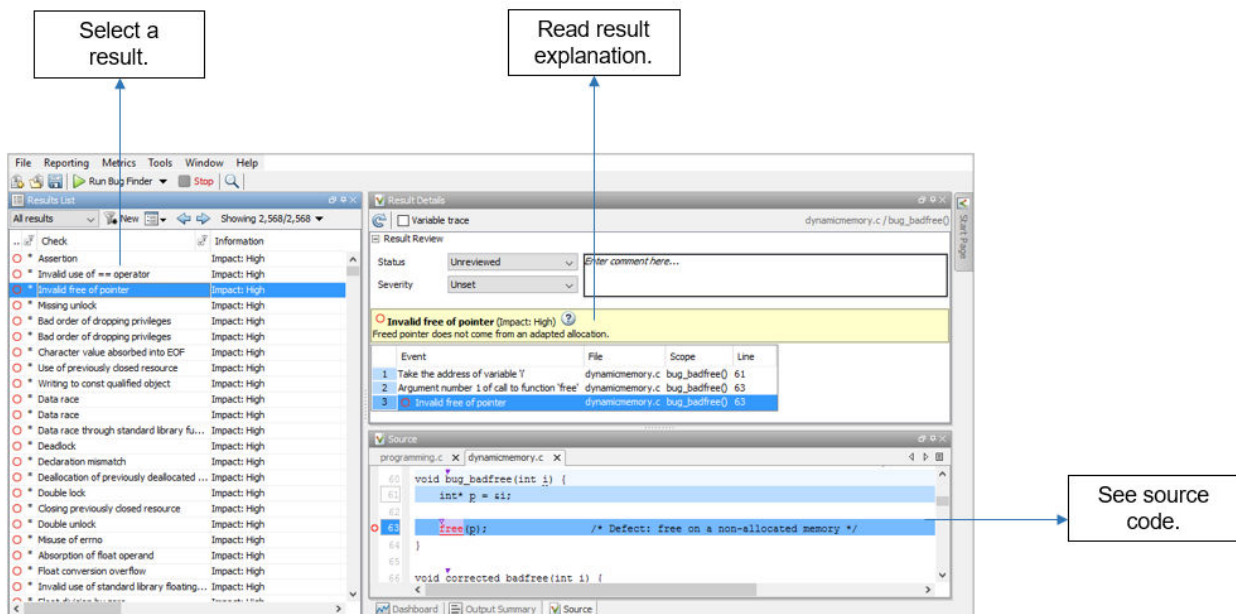
- For a similar workflow in the Polyspace Access web interface, see “Interpret Bug Finder Results in Polyspace Access Web Interface” on page 25-2.
- To see how to review results of Polyspace as You Code in IDEs, see “Review Polyspace as You Code Results in IDEs”.

When you open the results of a Polyspace Bug Finder analysis, you see a list on the **Results List** pane. The results consist of defects, coding rule violations or code metrics.

You can first narrow down the focus of your review:

- Use filters on the results list columns to narrow down the list. For instance, you can focus on the high-impact defects.
- Organize results by file or result family. Use the  icon above the list.

Once you narrow down the list, you can begin reviewing individual results. This topic describes how to review a result.



To begin your review, select a result in the list.

Interpret Result Details Message

The screenshot displays the Polyspace Desktop User Interface with a bug result highlighted. The **Result Details** pane shows the following information:

- Result ID:** ID 1740: Invalid use of == operator (Impact: High)
- Description:** Use of "==" instead of "=" in a statement or missing parentheses.
- Event Table:**

Event	File	Scope	Line
1	Invalid use of == operator	programming.c bug_badequalequaluse()	107

The **Source** pane shows the following code snippet:

```


100  /* BAD EQUAL=EQUAL USE
101  *
102  void bug_badequalequaluse(void) {
103      int i = 0;
104      int j = 5;
105      int arr[SIZE4];
106
107      for (i == 5; j < (SIZE4 + 5); j++) { /* Defect: Possible incorrect comp
108          arr[i] = j;
109          i++;
110      }
111  }
112  
```

The **Help** pane provides a detailed explanation of the error, including examples of correct and incorrect usage.

Interpret Message

The first step is to understand what is wrong. Read the message on the **Result Details** pane and the related line of code on the **Source** pane.

Seek Additional Resources for Help

Sometimes, you need additional help for certain results. Click the  icon to open a help page for the selected result. See code examples illustrating the result. Check external standards such as CWE or CERT-C that provide additional rationale for fixing the issue.

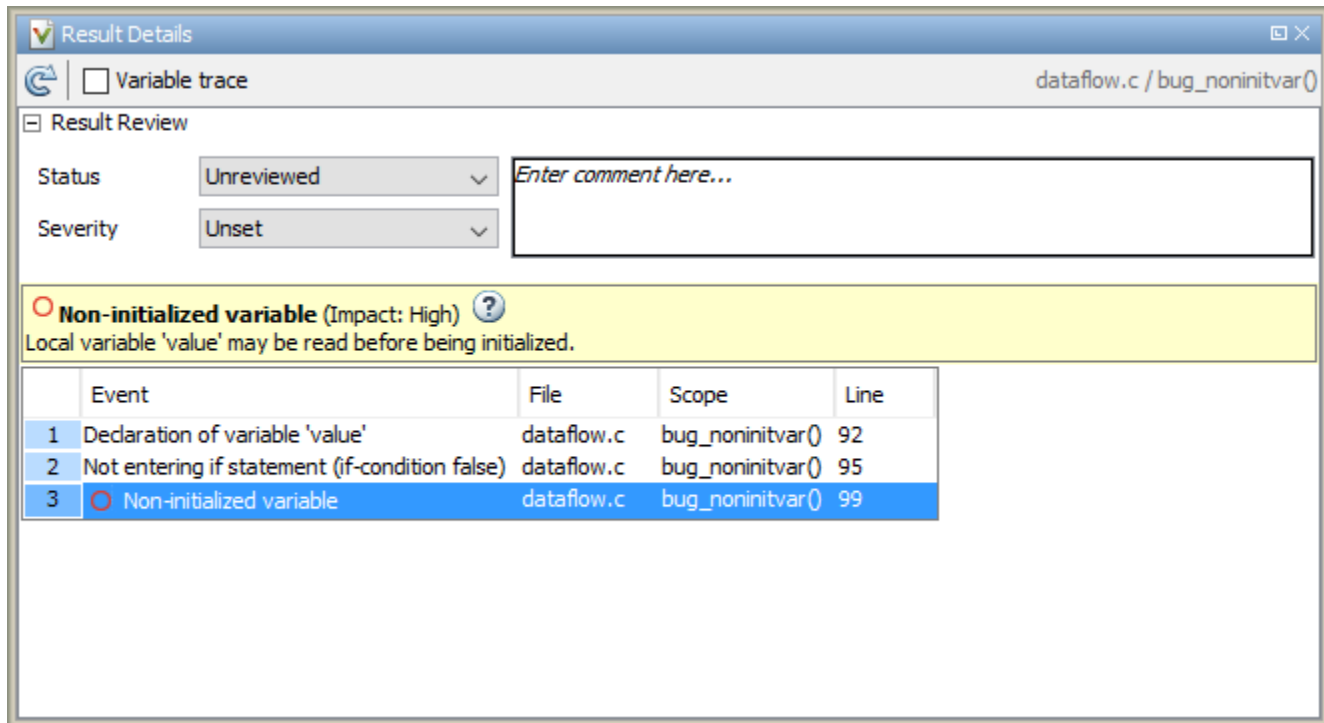
At this point, you might be ready to decide whether to fix the issue or not. Once you identify a fix, it might help to review all results of that type together.

Find Root Cause of Result

Sometimes, the root cause might be far from the actual location where the result is displayed. For instance, a variable that you read might be non-initialized because the initialization is not reachable. The defect is shown when you read the variable, but the root cause is perhaps a previous `if` or `while` condition that is always false.

Navigate to Related Events

Typically, the **Result Details** pane shows one sequence of events that leads to the result. The **Source** pane also highlights these events.



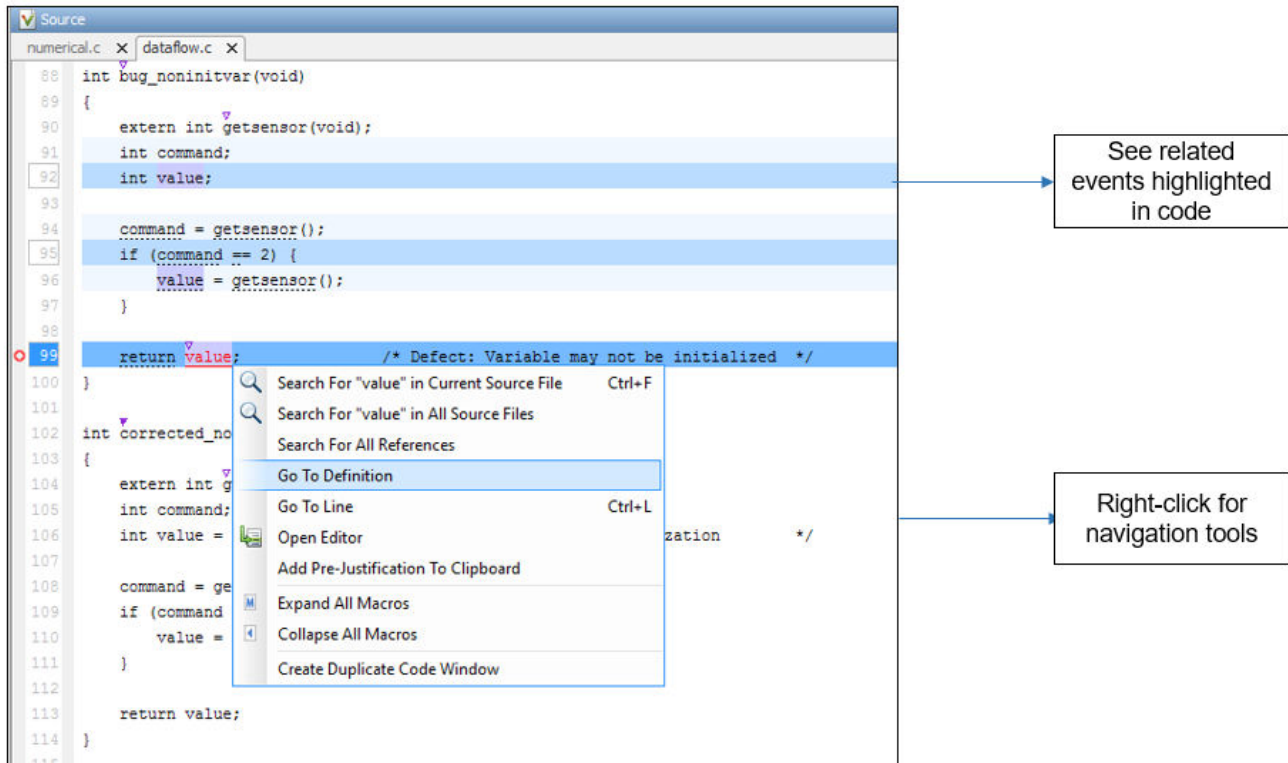
In the above event traceback, this sequence is shown:

- 1 A variable `value` is declared.
- 2 The execution path bypasses an `if` statement. This information might be relevant if the variable is initialized inside the `if` block.
- 3 Location of the current defect: **Non-initialized variable**

Typically, the traceback shows major points in the control flow: entering or bypassing conditional statements or loops, entering a function, and so on. For specific defects, the traceback shows other kinds of events relevant to the defect. For instance, for a **Declaration mismatch** defect, the traceback shows the two locations with conflicting declarations.

Create Your Own Navigation Path

If the event traceback is not available, use other navigation tools to trace your own path through the code.




Before you begin navigating through pathways in your code, ask the question: What am I looking for? Based on your answer, choose the appropriate navigation tool. For instance:

- To investigate a **Non-initialized variable** defect, you might want to make sure that the variable is not initialized at all. To look for previous instances of the variable, on the **Source** pane, right-click the variable and select **Search For All References**. Alternatively, double-click the variable. These options show only instances of a specific variable and not other variables with the same name in other scopes.
- To investigate a violation of **MISRA C:2012 Rule 17.7**:

The value returned by a function having non-void return type shall be used.

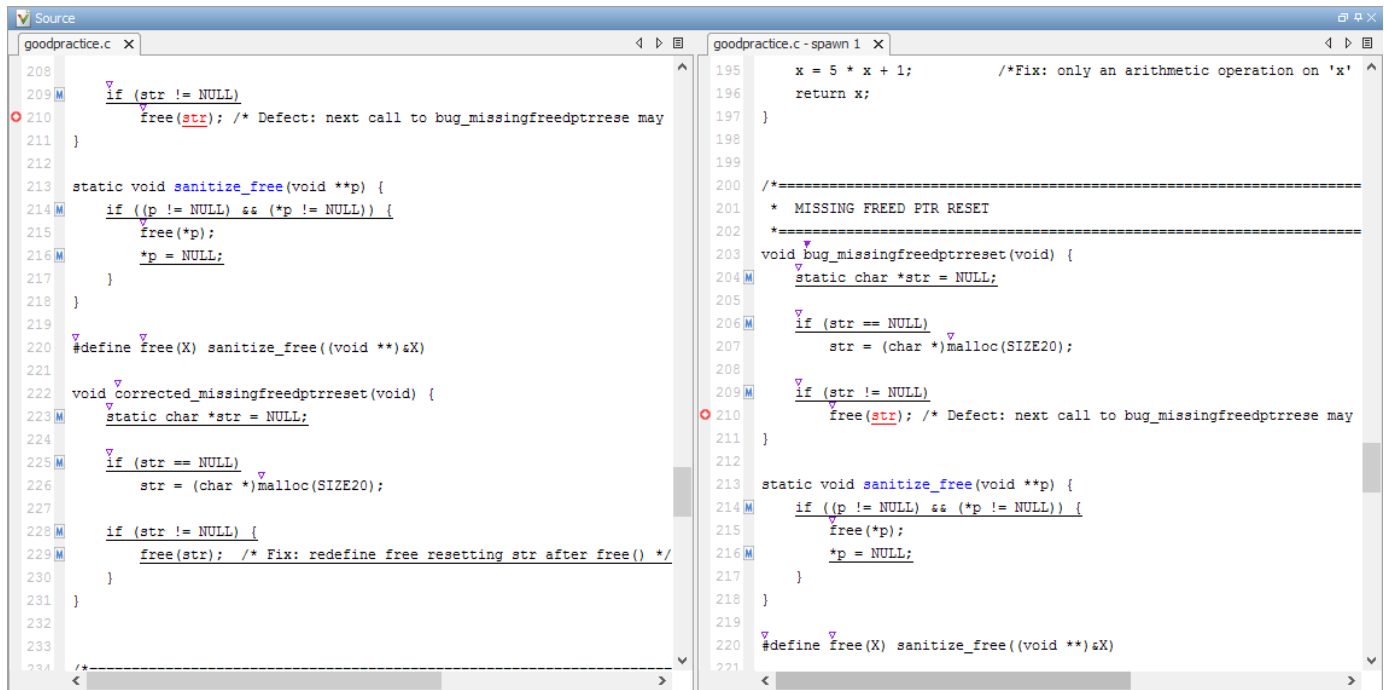
you might want to navigate from a function call to the function definition. Right-click the function and select **Go To Definition**.

After you navigate away from the current result, use the  icon on the **Result Details** pane to come back.

If you click a source code token containing a result, the previous result selection on the **Results List** and the details on the **Result Details** pane do not change. You can keep the result in the results list and the result details pinned while navigating in the source code. Sometimes, you might want to see the result associated with a token. To update the result selection and the details, Ctrl-click the token or right-click and select **Select Results At This Location**.

Navigate in Separate Window

If reviewing a result requires deeper navigation in your source code, you can create a duplicate source code window that focuses on the result while you navigate in the original source code window.



Right-click on the **Source** pane and select **Create Duplicate Code Window**. Right-click on the tab showing the duplicate file name (ending with - spawn 1) and select **New Vertical Group**.

Perform the navigation steps in the duplicate file window while the defect still appears on the original file window. After the investigation is over, close the duplicate window.

See Also

More About

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 22-2
- “Filter and Group Results in Polyspace Desktop User Interface” on page 23-2

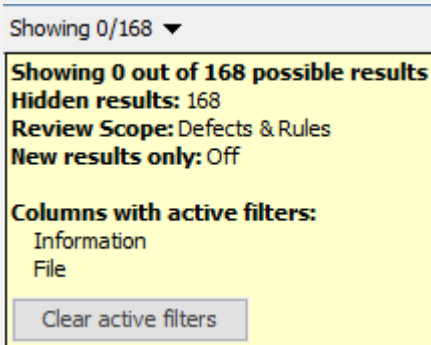
Investigate the Cause of Empty Results List

This topic shows how to interpret an empty results list in the user interface of the Polyspace desktop products. To see how to interpret a similar empty list in the Polyspace Access web interface, see “Investigate the Cause of Empty Results List” on page 25-7.

When you run an analysis with Polyspace Bug Finder, the **Results List** pane can be empty or it can display this message:

Polyspace Bug Finder did not find any defects or coding rule violations in your code.

The message can indicate that your code has no defect or coding rule violation. However, before you reach this conclusion, check the following.

Possible Cause	Action to Take
Did all your source files compile?	In the Output Summary pane, look for warning messages that start with: Failed compilation. If a file does not compile, Bug Finder can return some results, but only files with no compilation errors are fully analyzed.
Did you include all your source files in your project?	In the Project Browser pane, make sure that all the files that you want to analyze are included in the Project Source Files folder.
Did you configure your project correctly?	In the Configuration pane: <ul style="list-style-type: none"> Under Coding Standards & Code Metrics, verify that you have selected the appropriate options if you want to check Coding Rules and compute Code Metrics. Under Bug Finder Analysis, confirm that you have selected all the defects that you want to check during the analysis. Under Run Settings, see if Use fast analysis mode for Bug Finder is selected. In this mode, Bug Finder checks for only a subset of defects and coding rules.
Are you applying any filters to the results?	In the Results List pane header, make sure that there are no Hidden results in the Showing drop-down list. To clear all applied filters, click Clear active filters .  <p>The screenshot shows a yellow box with the following text: "Showing 0/168" with a dropdown arrow, "Showing 0 out of 168 possible results", "Hidden results: 168", "Review Scope: Defects & Rules", "New results only: Off", "Columns with active filters:" followed by "Information" and "File", and a "Clear active filters" button.</p>

See Also

More About

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 22-2
- “Troubleshoot Compilation Errors”

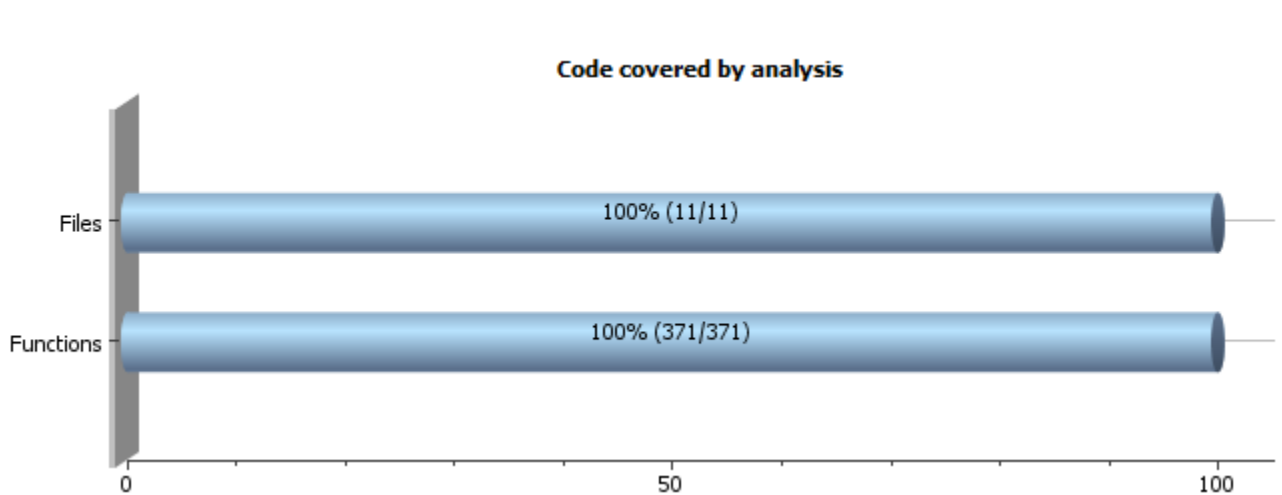
Dashboard in Polyspace Desktop User Interface

This topic focuses on the Polyspace desktop user interface. To learn about the equivalent pane in the Polyspace Access web interface, see “Dashboard in Polyspace Access Web Interface” on page 25-9.

The **Dashboard** pane provides statistics on the analysis results in a graphical format.

When you open a results file in Polyspace, this pane is displayed by default. You can view the following graphs:

Code Covered by Analysis



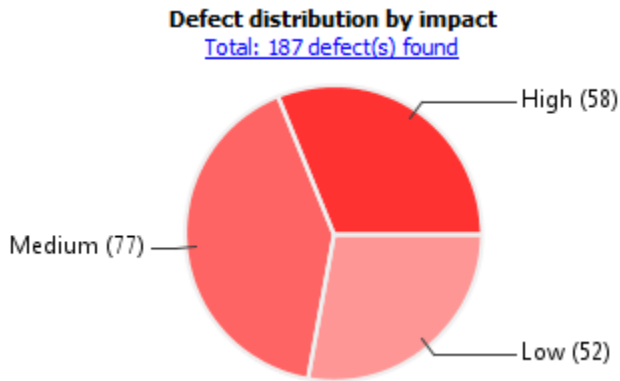
From this graph you can obtain the following information:

- **Files:** Ratio of analyzed files to total number of files. If a file contains a compilation error, Polyspace Bug Finder does not analyze the file.

If some of your files were only partially analyzed because of compilation errors, this pane contains a link stating that some files failed to compile. To see the compilation errors, click the link and navigate to the **Output Summary** pane.

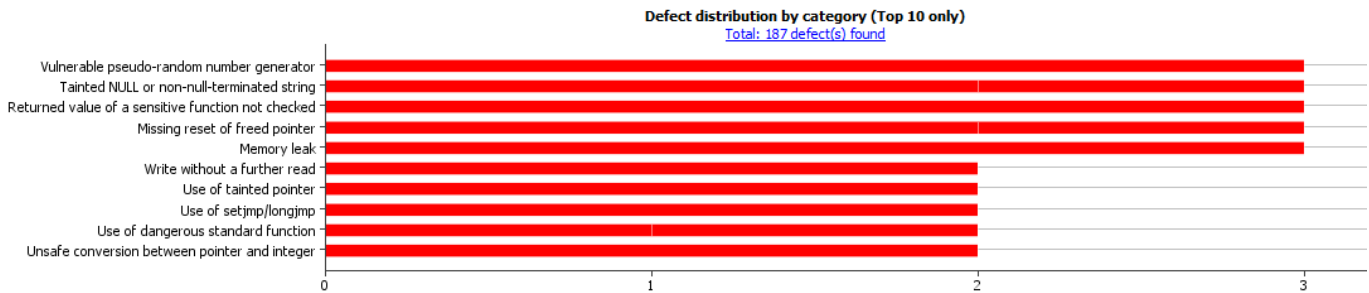
- **Functions:** Ratio of analyzed functions to total number of functions *in the analyzed files*. If the analysis of a function takes longer than a certain threshold value, Polyspace Bug Finder does not analyze the function.

Defect Distribution by Impact



From this pie chart, you can obtain a graphical visualization of the defect distribution by impact. You can find at a glance whether the defects that Polyspace Bug Finder found in your code are low-impact defects. For more information on impact, see “Classification of Defects by Impact” on page 18-49.

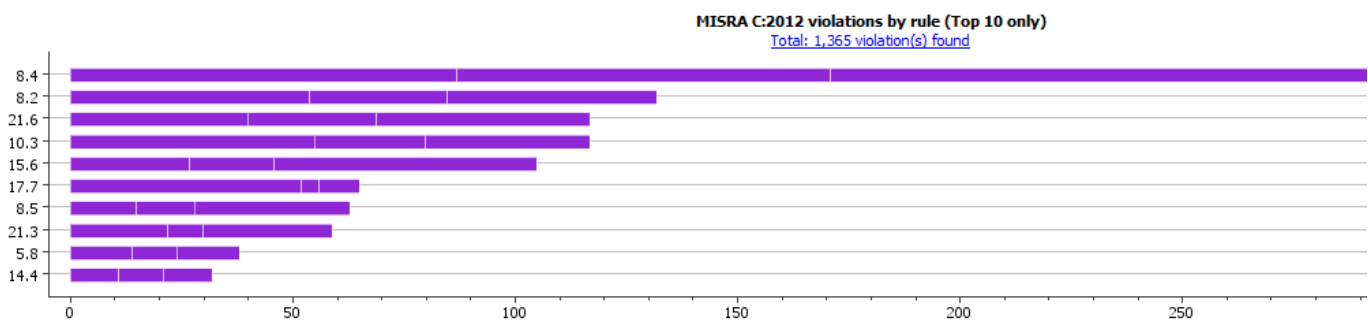
Defect Distribution by Category or File



From this graph you can obtain the following information.

	Category	File
Top 10	<p>The ten defect types with the highest number of individual defects.</p> <ul style="list-style-type: none"> Each column represents a defect type and is divided into the: <ul style="list-style-type: none"> File with highest number of defects of this type. File with second highest number of defects of this type. All other files with defects of this type. <p>Place your cursor on a column to see the file name and number of defects of this type in this file.</p> <ul style="list-style-type: none"> The x-axis represents the number of defects. <p>Use this view to organize your check review starting at defect types with more individual defects.</p>	<p>The ten source files with the highest number of defects.</p> <ul style="list-style-type: none"> Each column represents a file and is divided into the: <ul style="list-style-type: none"> Defect type with highest number of defects in this file. Defect type with second highest number of defects in this file. All other defect types in this file. <p>Place your cursor on a column to see the defect type name and number of defects of this type in this file.</p> <ul style="list-style-type: none"> The x-axis represents the number of defects. <p>Use this view to organize your check review starting at files with more defects.</p>
Bottom 10	<p>The ten defect types with the lowest number of individual defects. Each column on the graph is divided the same way as the Top 10 defect types.</p> <p>Use this view to organize your check review starting at defect types with fewer individual defects.</p>	<p>The ten source files with the lowest number of defects. Each column on the graph is divided the same way as the Top 10 files.</p> <p>Use this view to organize your check review starting at files with fewer defects.</p>

Coding Rule Violations by Rule or File



For every type of coding rule that you check (MISRA, JSE, or custom), the **Dashboard** contains a graph of the rule violations.

From this graph you can obtain the following information.

	Category	File
Top 10	<p>The ten rules with the highest number of violations.</p> <ul style="list-style-type: none"> Each column represents a rule number and is divided into the: <ul style="list-style-type: none"> File with highest number of violations of this rule. File with second highest number of violations of this rule. All other files with violations of this rule. <p>Place your cursor on a column to see the file name and number of violations of this rule in the file.</p> <ul style="list-style-type: none"> The x-axis represents the number of rule violations. <p>Use this view to organize your review starting at rules with more violations.</p>	<p>The ten source files containing the highest number of violations.</p> <ul style="list-style-type: none"> Each column represents a file and is divided into the: <ul style="list-style-type: none"> Rule with highest number of violations in this file. Rule with second highest number of violations in this file. All other rules violated in this file. <p>Place your cursor on a column to see the rule number and number of violations of the rule in this file.</p> <ul style="list-style-type: none"> The x-axis represents the number of rule violations. <p>Use this view to organize your review starting at files with more rule violations.</p>
Bottom 10	<p>The ten rules with the lowest number of violations. Each column on the graph is divided in the same way as the Top 10 rules.</p> <p>Use this view to organize your review starting at rules with fewer violations.</p>	<p>The ten source files containing the lowest number of rule violations. Each column on the graph is divided in the same way as the Top 10 files.</p> <p>Use this view to organize your review starting at files with fewer rule violations.</p>

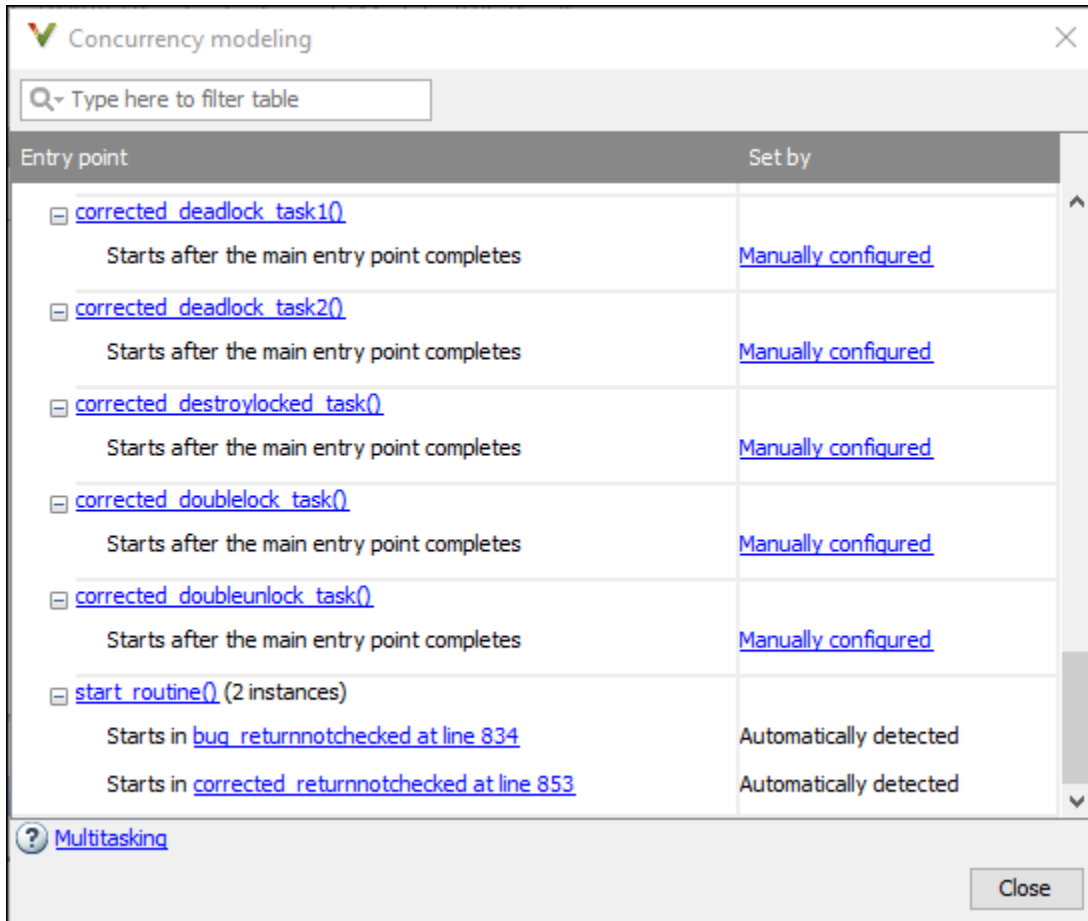
Other Dashboard Features

You can also perform the following actions on this pane:

- Select elements on the graphs to filter results from the **Results List** pane. See “Filter and Group Results in Polyspace Desktop User Interface” on page 23-2.
- View the configuration used to obtain the result. Select the link **Configuration**.
- View the modeling of the multitasking configuration of your code. Select the link **Concurrency modeling on page 21-13**.

Concurrency Modeling in Polyspace Desktop User Interface

The **Concurrency Modeling** view displays all the tasks and interrupts that the analysis extracts from your code and your Polyspace multitasking configuration.



in the table, the functions are listed in the first column by order of decreasing priority. The second column shows how Polyspace detects each task or interrupt: automatically, manually from the Polyspace configuration, or from an external file.

From this view, you can:

- Click a function name to go to its definition in the source code.
- Click an event to go to the corresponding call to the concurrency primitive in the source code, for instance `pthread_create`.
- Click **Manually configured**, for functions that are manually configured, to go to the **Multitasking** node on the **Configuration** pane.

See Also


External multitasking configuration

More About

- “Analyze Multitasking Programs in Polyspace” on page 15-2
- “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 15-7
- “Configuring Polyspace Multitasking Analysis Manually” on page 15-17

Results List in Polyspace Desktop User Interface

This topic focuses on the Polyspace desktop user interface. To learn about the equivalent pane in the Polyspace Access web interface, see “Results List in Polyspace Access Web Interface” on page 25-19.

The **Results List** pane lists all results along with their attributes. To organize your results review, from the  list on this pane, select one of the following options:

- **None:** Lists defects and coding rule violations without grouping. By default the results are listed in order of severity.
- **Family:** Lists results grouped by grouping. For more information on the defects covered by a group, see “Bug Finder Defect Groups” on page 18-43.
- **Class:** Lists results grouped by class. Within each class, the results are grouped by method. The first group, **Global Scope**, lists results not occurring in a class definition.

This option is available for C++ code only.

- **File:** Lists results grouped by file. Within each file, the results are grouped by function.

For each result, the **Results List** pane contains the result attributes, listed in columns:

Attribute	Description
Family	Group to which the result belongs.
ID	Unique identification number of the result.
Type	Defect or coding rule violation.
Group	Category of the result, for instance: <ul style="list-style-type: none"> • For defects: Groups such as static memory, numerical, control flow, concurrency, etc. • For coding rule violations: Groups defined by the coding rule standard. For instance, MISRA C:2012 defines groups related to code constructs such as functions, pointers and arrays, etc.
Check	Result name, for instance: <ul style="list-style-type: none"> • For defects: Defect name • For coding rule violations: Coding rule number
Detail	Additional information about a result. The column shows the first line of the Result Details pane. For an example of how to use this column, see the result MISRA C:2012 Dir 1.1.
File	File containing the instruction where the result occurs
Class	Class containing the instruction where the result occurs. If the result is not inside a class definition, then this column contains the entry, Global Scope .
Function	Function containing the instruction where the result occurs. If the function is a method of a class, it appears in the format <code>class_name::function_name</code> .
Folder	Path to the folder that contains the source file with the result

Attribute	Description
CWE ID	<p>CWE IDs corresponding to Bug Finder results. See “Extend CWE Coding Standard Coverage Using Polyspace Defect Checkers” on page 18-97.</p> <hr/> <p>Note This column is available only for results that were generated with a Polyspace product version R2022b or earlier. For R2023a and later versions, Polyspace shows CWE violations in same way as violations of other coding standards. See “Common Weakness Enumeration (CWE)”.</p>
Severity	<p>Level of severity you have assigned to the result. The possible levels are:</p> <ul style="list-style-type: none"> • Unset • High • Medium • Low
Status	<p>Review status you have assigned to the result. The possible statuses are:</p> <ul style="list-style-type: none"> • Unreviewed (default status) • To investigate • To fix • Justified • No action planned • Not a defect • Other <p>To create and use custom review statuses, see “Create Custom Review Status” on page 2-24.</p>
Comments	<p>Comments you have entered about the result</p>
Assigned to	<p>User name of reviewer assigned to this result.</p> <p>This column is visible only for results that you open from Polyspace Access.</p>
Ticket Key	<p>When you create a bug tracking tool (BTT) ticket for a result, this field contains the ticket ID. Click the ticket ID in the Results Details to open the ticket in the BTT interface.</p> <p>This column is visible only for results that you open from Polyspace Access.</p>


To show or hide any of the columns, right-click anywhere on the column titles. From the context menu, select or clear the title of the column that you want to show or hide.

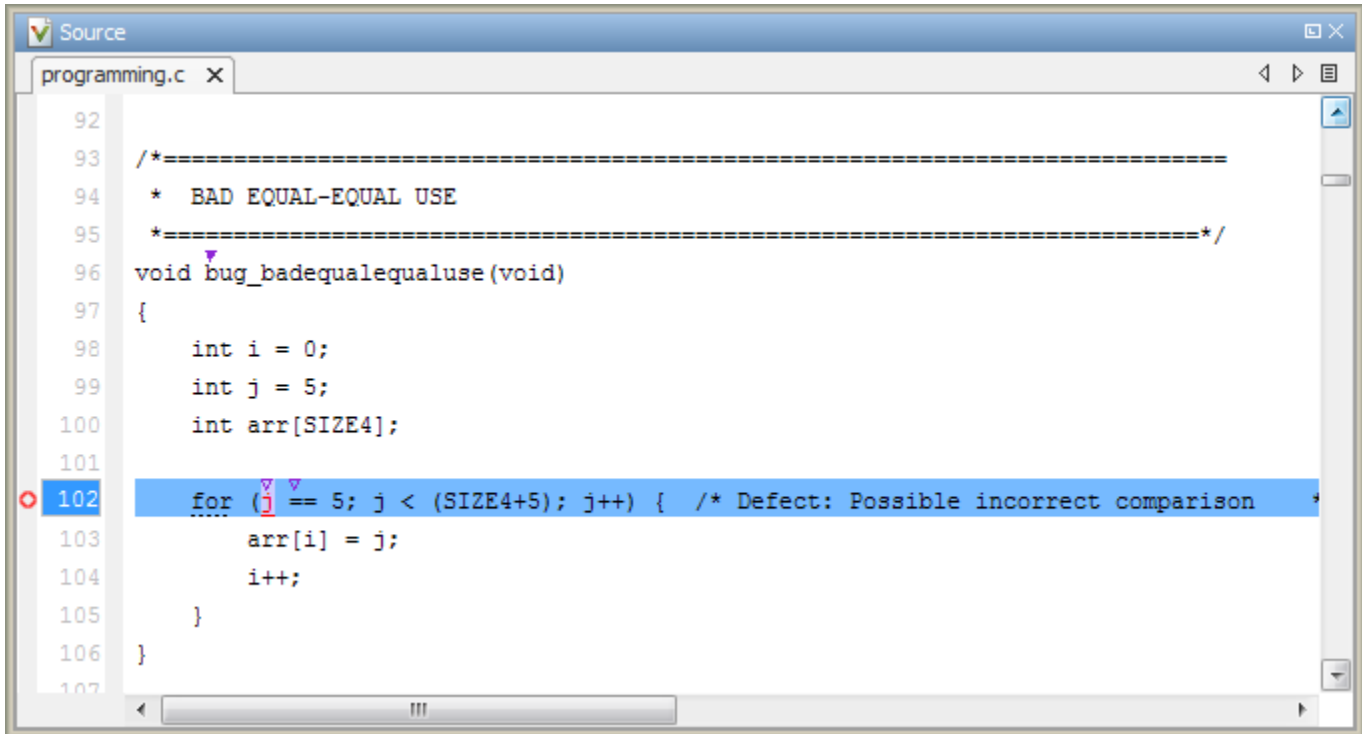
Using this pane, you can:

- Navigate through the results.
- Organize your result review using filters on the columns. For more information, see “Filter and Group Results in Polyspace Desktop User Interface” on page 23-2.

Source Code in Polyspace Desktop User Interface

This topic focuses on the Polyspace desktop user interface. To learn about the equivalent pane in the Polyspace Access web interface, see “Source Code in Polyspace Access Web Interface” on page 25-21.

The **Source** pane shows the source code with the defects colored in red and the corresponding line number marked by .



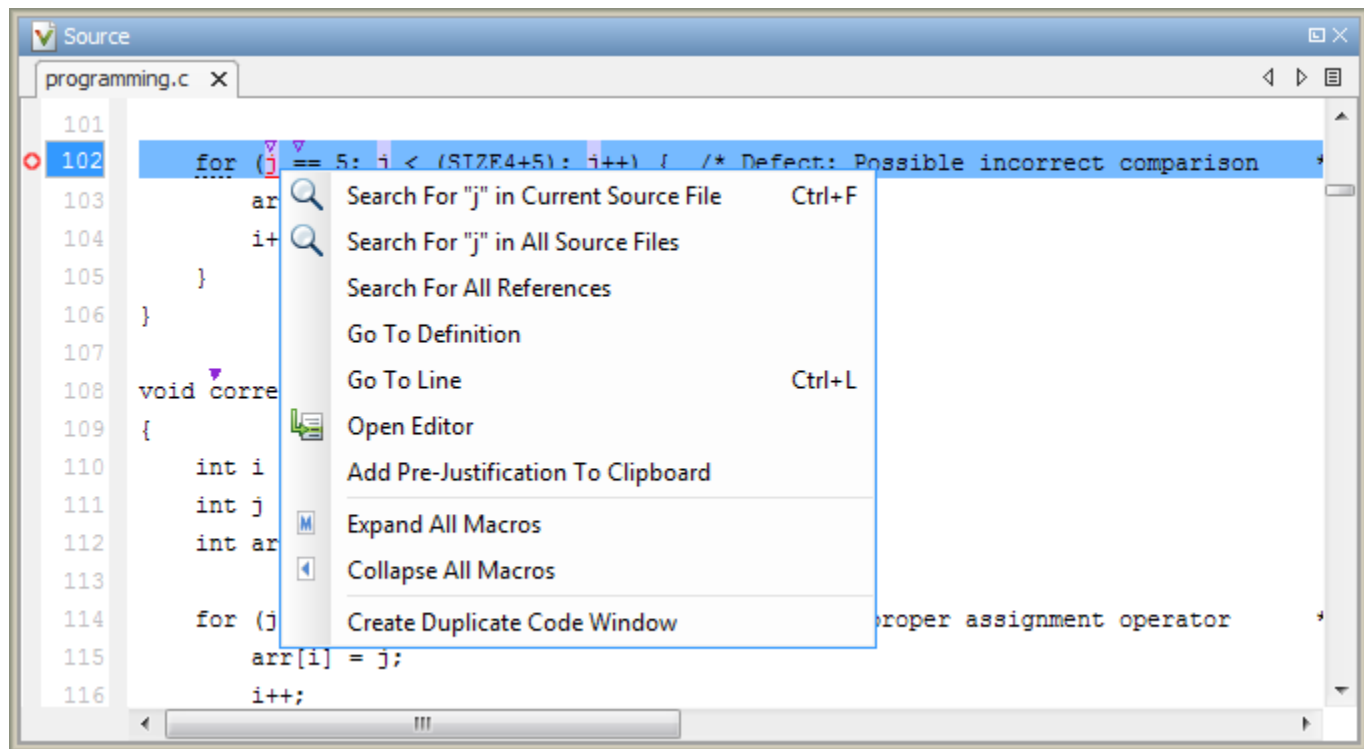
```

92
93 /*=====
94  *  BAD EQUAL-EQUAL USE
95  *=====*/
96 void bug_badequalequaluse(void)
97 {
98     int i = 0;
99     int j = 5;
100    int arr[SIZE4];
101
102    for (j == 5; j < (SIZE4+5); j++) { /* Defect: Possible incorrect comparison
103        arr[i] = j;
104        i++;
105    }
106 }
107
  
```

Placing your cursor over a result displays a tooltip that provides range information for variables, operands, function parameters, and return values.

Examine Source Code


On the **Source** pane, if you right-click a text string, the context menu provides options to examine your code:

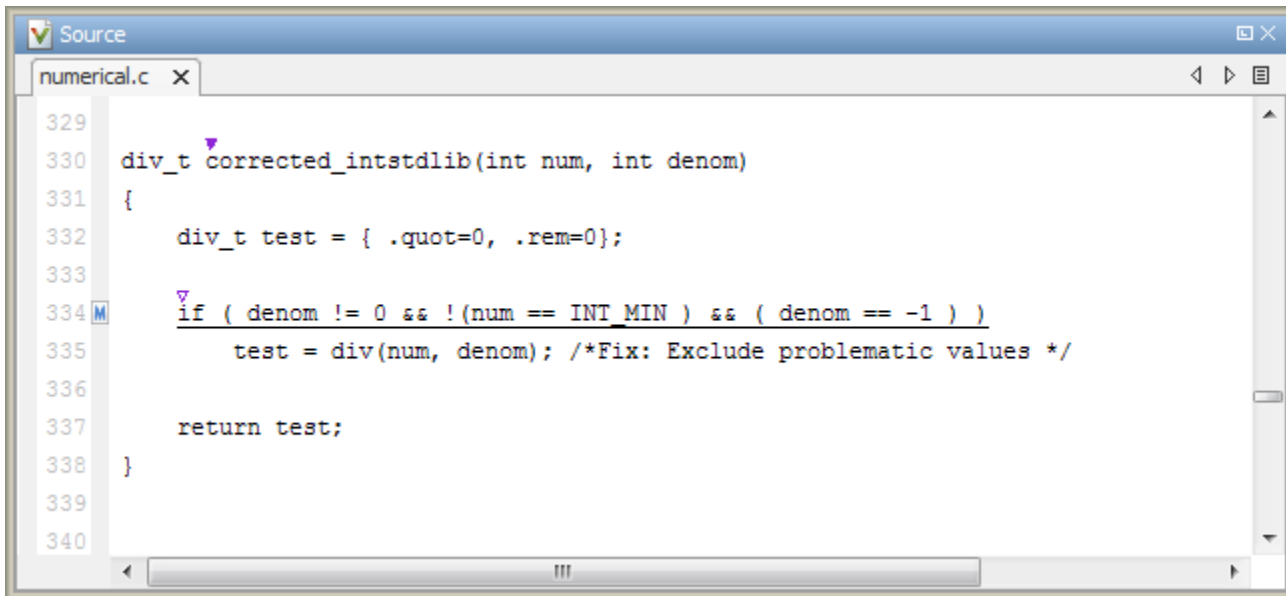


For example, if you right-click the variable `i`, you can use the following options to examine and navigate through your code:

- **Search "j" in Current Source** — List occurrences of the string within the current source file on the **Search** pane.
- **Search "j" in All Source Files** — List occurrences of the string within the source files on the **Search** pane.
- **Search For All References** — List all references in the **Search** pane. The software supports this feature for global and local variables, functions, types, and classes.
- **Go To Definition** — Go to the line of code that contains the definition of `i`. The software supports this feature for global and local variables, functions, types, and classes. If a definition is not available to Polyspace, selecting the option takes you to the declaration.
- **Go To Line** — Open the Go to line dialog box. If you specify a line number and click **Enter**, the software displays the specified line of code.
- **Expand All Macros** or **Collapse All Macros** — Display or hide the content of macros in current source file.

Expand Macros

You can view the contents of source code macros in the source code view. A code information bar displays  icons that identify source code lines with macros.

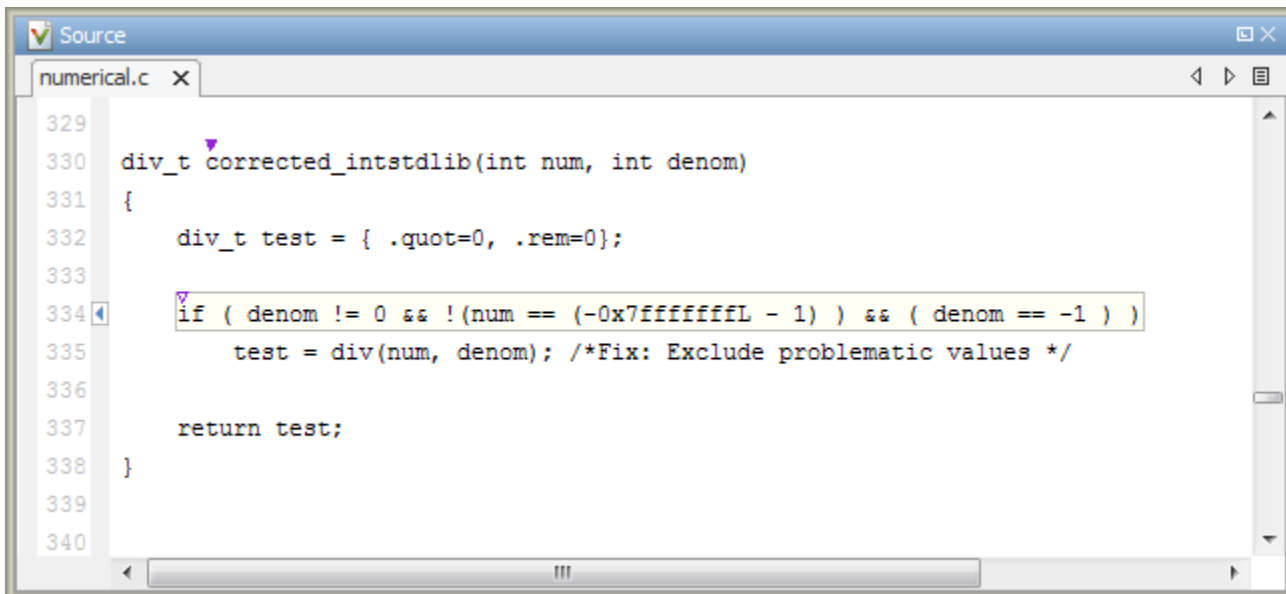


```

329
330 div_t corrected_intstdlib(int num, int denom)
331 {
332     div_t test = { .quot=0, .rem=0};
333
334     if ( denom != 0 && !(num == INT_MIN ) && ( denom == -1 ) )
335         test = div(num, denom); /*Fix: Exclude problematic values */
336
337     return test;
338 }
339
340

```


When you click a line with this icon, the software displays the contents of macros on that line in a box.



```

329
330 div_t corrected_intstdlib(int num, int denom)
331 {
332     div_t test = { .quot=0, .rem=0};
333
334     if ( denom != 0 && !(num == (-0x7fffffffL - 1) ) && ( denom == -1 ) )
335         test = div(num, denom); /*Fix: Exclude problematic values */
336
337     return test;
338 }
339
340

```

To display the normal source code again, click the line away from the box, for example, on the  icon.

To display or hide the content of *all* macros:

- 1 Right-click anywhere on the source.
- 2 From the context menu, select either **Expand All Macros** or **Collapse All Macros**.

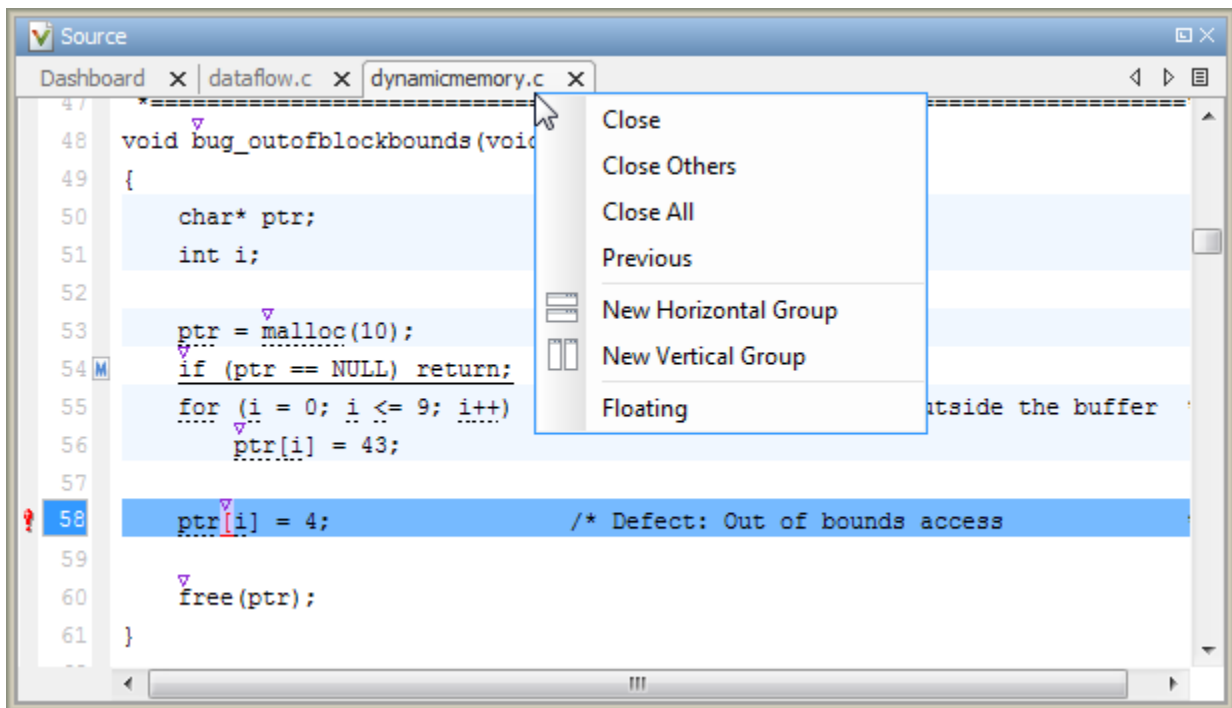
Note

- 1 The **Result Details** pane also allows you to view the contents of a macro if the check you select lies within a macro.
- 2 You cannot expand OSEK API macros in the **Source** pane.

Manage Multiple Files in Source Pane

You can view multiple source files in the **Source** pane.

Right-click on the **Source** pane toolbar.

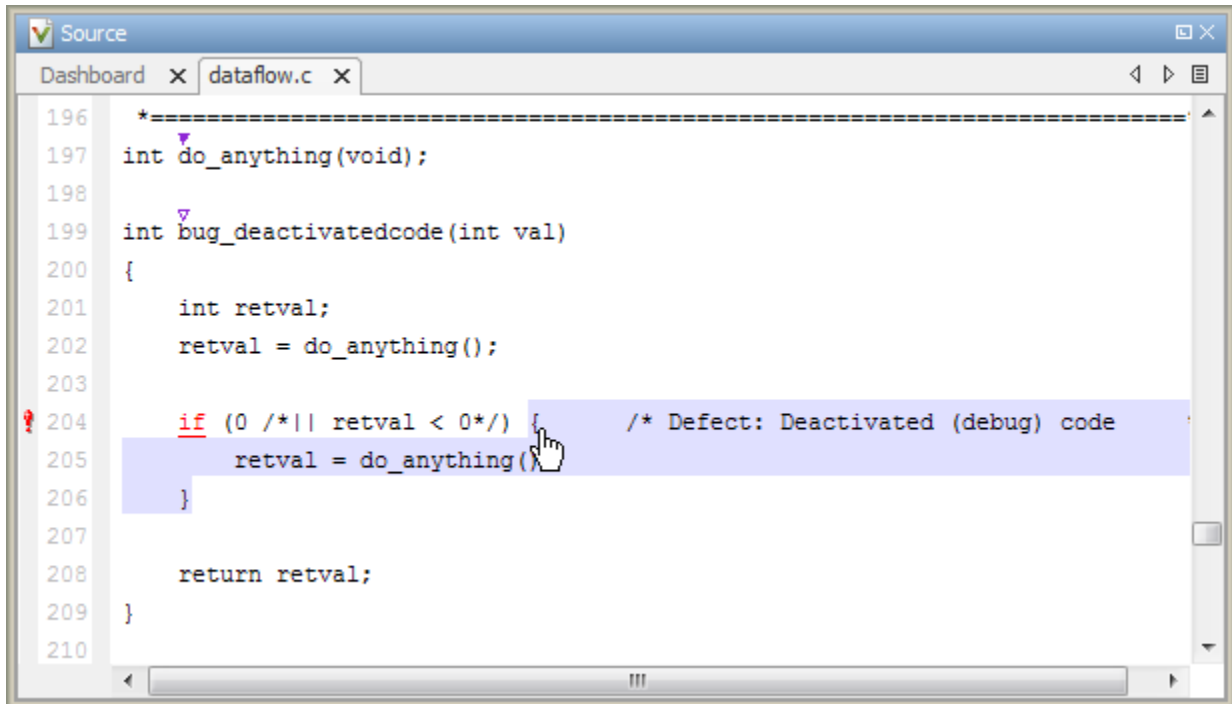


From the **Source** pane context menu, you can:

- **Close** - Close the currently selected source file. You can also use the χ button to close tabs.
- **Close Others** - Close all source files except the currently selected file.
- **Close All** - Close all source files.
- **Next** - Display the next view.
- **Previous** - Display the previous view.
- **New Horizontal Group** - Split the Source window horizontally to display the selected source file below another file.
- **New Vertical Group** - Split the Source window vertically to display the selected source file side-by-side with another file.
- **Floating** - Display the current source file in a new window, outside the **Source** pane.

View Code Block

On the **Source** pane, to highlight a block of code, click either its opening or closing brace. If the brace itself is highlighted, click the brace twice.

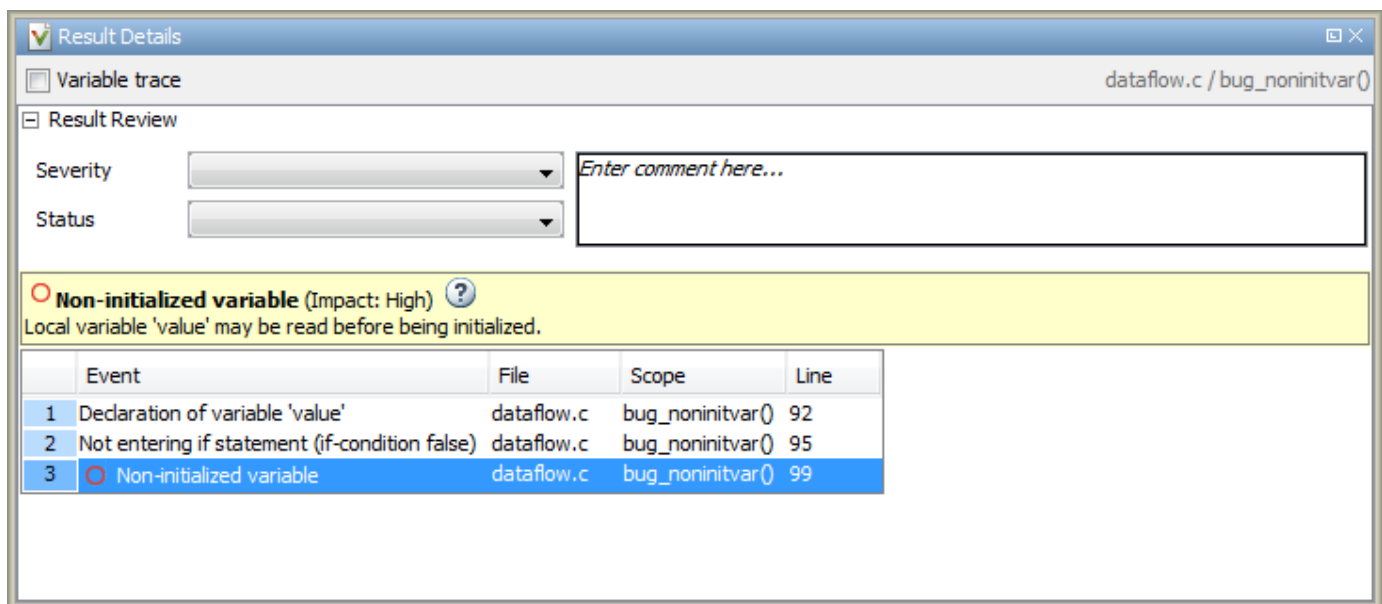



Result Details in Polyspace Desktop User Interface

This topic focuses on the Polyspace desktop user interface. To learn about the equivalent pane in the Polyspace Access web interface, see “Result Details in Polyspace Access Web Interface” on page 25-26.

The **Result Details** pane contains comprehensive information about a specific defect. To see this information, on the **Results List** pane, select the defect.

On this pane, you can also assign a **Severity** and **Status** to each check. You can also enter comments to describe the results of your review. This action helps you track the progress of your review and avoid reviewing the same check twice.




- The top right corner shows the file and function containing the defect, in the format *file_name/function_name*.
- The yellow box contains the name of the defect with an explanation of why the defect occurs.
- The **Event** column lists the sequence of code instructions causing the defect. The **Scope** column lists the function containing the instructions. If the instructions are not in a function, the column lists the file containing the instructions. The **Line** column lists the line number of the instructions.
- The **Variable trace** check box allows you to see an additional set of instructions that are related to the defect.
- The  button allows you to access documentation for the defect.
- For results that you open from Polyspace Access, you can also:
 - Assign a reviewer to the result. A reviewer can filter the **Results List** to only show results that are assigned to him or her.
 - Create a ticket in a bug tracking tool (BTT) such as JIRA. Once you create the ticket the **Results Details** for this defect shows the ticket ID. Click the ID to open the ticket in the BTT interface.


See “Open or Export Results from Polyspace Access” on page 28-2.

Call Hierarchy in Polyspace Desktop User Interface

This topic focuses on the Polyspace desktop user interface. To learn about the equivalent pane in the Polyspace Access web interface, see “Call Hierarchy in Polyspace Access Web Interface” on page 25-28.

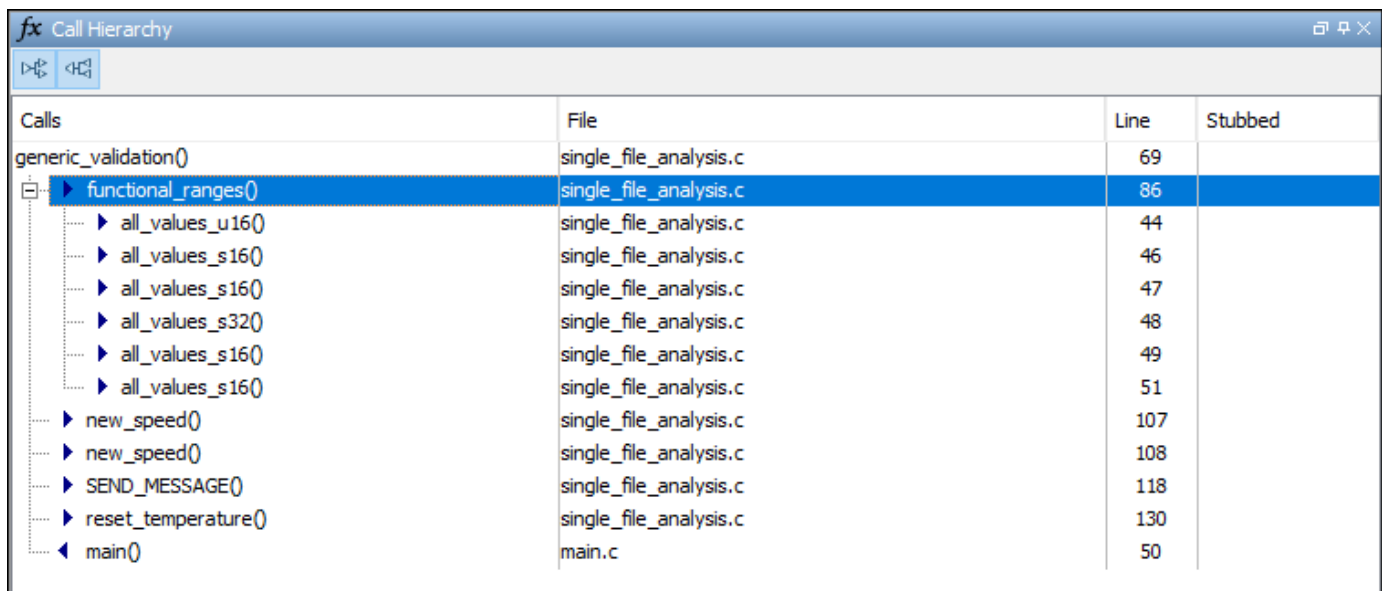
The **Call Hierarchy** pane displays the call tree of functions in the source code.

For each function `foo`, the **Call Hierarchy** pane lists the functions and tasks that call `foo` (callers) and those called by `foo` (callees). The callers are indicated by ◀ (functions) or ◀|| (tasks). The callees are indicated by ▶ (functions) or ||▶ (tasks). The **Call Hierarchy** pane lists direct function calls and indirect calls through function pointers. The indirect calls are shown with the  icon.

You open the **Call Hierarchy** pane by using the  icon in your result details. To update the pane:

- You can click a defect, either on the **Results List** or **Source** pane. You see the function containing the defect with its callers and callees.
- You can right-click the name of a function and select **Go To Definition**. You see the callers and callees of the function.

In this example, the **Call Hierarchy** pane displays the function `generic_validation`, and with its callers and callees.



Calls	File	Line	Stubbed
generic_validation()	single_file_analysis.c	69	
▶ functional_ranges()	single_file_analysis.c	86	
▶ all_values_u16()	single_file_analysis.c	44	
▶ all_values_s16()	single_file_analysis.c	46	
▶ all_values_s16()	single_file_analysis.c	47	
▶ all_values_s32()	single_file_analysis.c	48	
▶ all_values_s16()	single_file_analysis.c	49	
▶ all_values_s16()	single_file_analysis.c	51	
▶ new_speed()	single_file_analysis.c	107	
▶ new_speed()	single_file_analysis.c	108	
▶ SEND_MESSAGE()	single_file_analysis.c	118	
▶ reset_temperature()	single_file_analysis.c	130	
◀ main()	main.c	50	

The line number in the **Call Hierarchy** pane refers to a different line in the source code:

- For the function name, the line number refers to the beginning of the function definition. The definition of `generic_validation` begins on line 69.
- For a callee name, the number refers to the line where the callee is called. The callee `functional_ranges` is called by `generic_validation` on line 86.
- For a caller name, the number refers to the line where the caller calls the function. The caller `main` calls `generic_validation` on line 50.

Tip To navigate to the call location in the source code, select a caller or callee name

Actions Supported on Call Hierarchy Pane

In the **Call Hierarchy** pane, you can perform these actions.

Show or Hide Callers and Callees

Customize the view to display callers only or callees only. Show or hide callers and callees by clicking this button



Navigate Call Hierarchy

You can navigate the call hierarchy in your source code. For a function, double-click a caller or callee name to navigate to the caller or callee definition in the source code.

Determine if Function is Stubbed

From the **Stubbed** column, you can determine if a function is stubbed. The entries in the column show why a function was stubbed.

- **Automatic:** Polyspace cannot find the function definition. For instance, you did not provide the file containing the definition.
- **Std library:** The function is a standard library function. You do not provide the function definition explicitly in your Polyspace project.
- **Mapped to std library:** You map the function to a standard library function by using the option `-code-behavior-specifications`.

Limitations of Call Hierarchy Display in Bug Finder

In Polyspace Bug Finder, you might not see all callers or callees of a function, especially for calls through function pointers and dead code.

For instance, the **Call Hierarchy** pane:

- Does not display the functions registered with `at_exit()` and `at_quick_exit()`, and called by `exit()` and `quick_exit()` respectively.
- Might not show all function calls if the calls are made through a large array of function pointers.
- Might not show all calls to C++ `virtual` functions.

Understanding Changes in Polyspace Results After Product Upgrade

This topic describes how to interpret changes in results after upgrading Polyspace Code Prover. For product upgrade instructions, see “Update Polyspace Products” (Polyspace Code Prover) or “Update or Uninstall Polyspace Access” (Polyspace Code Prover).

If you upgrade to a newer release of Polyspace, you can see some changes in results for the same analysis. Each release introduces many improvements in analysis precision. These improvements can lead to the same analysis (same source files and options) showing a difference in results before and after the upgrade.

This topic describes the kinds of differences you might see, why they might be expected and how you can understand those differences. For information on how to compare two sets of results, see “Migrate Polyspace Projects After Product Upgrade”.

Changes in Polyspace Code Prover Results

For the same source code and analysis configuration, you might see a change in results because of improvements to the Polyspace Code Prover analysis engine. In Code Prover, a change in result means a change in color for the same run-time check. When comparing results, you can focus only on new red, grey and orange checks. As explained later, new green checks are typically the result of an increase in precision.

- For major differences in results of a specific type, see if you can trace the difference to a documented change in behavior or assumptions.

Check the release notes of all releases between your prior release and the release you upgraded to. Look in the **Verification results** section in the *Release Notes for Polyspace Code Prover* (Polyspace Code Prover) for changes in behavior of specific checks or changes in Code Prover assumptions. Major changes in behavior or assumptions are typically documented in the release notes.

- For differences in results that cannot be traced to a documented change, see if you can attribute the change in color to an increase in verification precision.

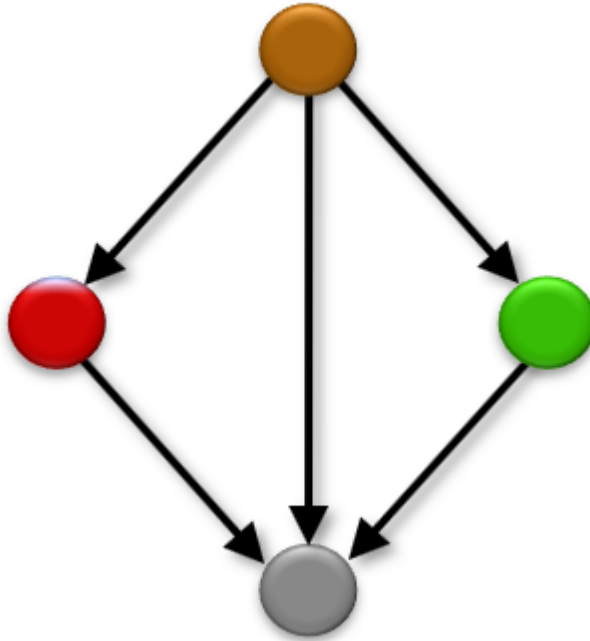
In addition to documented major changes, each new release also involves many minor improvements in the verification algorithms. These improvements typically lead to an increase in verification precision (or at least maintain the same precision as before).

Run-time checks in Polyspace Code Prover can return results in one of three colors:

- Red (proof of definite error)
- Green (proof of definite absence of error)
- Red (unproven, probable error)

In addition, a grey color is used for code that is unreachable and therefore not checked for other run-time errors. For more details, see “Code Prover Result and Source Code Colors” (Polyspace Code Prover).

An increase in precision indicates changes in color in the direction shown in this figure:



In other words, one of these things might happen:

- **Orange to green or red:** An operation that shows an orange color might now be red or green. Code Prover has a more precise knowledge of variable ranges when analyzing that operation, so the presence or absence of an error can be proven.

During verification, to check if an operation causes run-time errors, Code Prover uses previously computed possible values of the operands. Some of these values might be accumulated from the code (from initializations and subsequent write operations along various paths), but some of them are results of approximations and cannot happen in practice. An increase in precision leads to fewer approximations, so fewer values that come solely from approximations. As a result, it is possible to obtain proof in a greater number of cases.

Consider this simple, illustrative example:

```

x = initialValue;
y = x - 1;

// Call below increases its second arg if the first is positive
incr_y_if_x_positive(x,&y);

// Later operations involving x and y
interval = x - y ;
if (x > 0)
    interval = x;
num = range / interval;
  
```

In this example, to prove that there is no division by zero, Code Prover has to keep track of the fact that `interval = x - y` can be zero only if `x > 0` (and the fact that the case `x > 0` is handled later). Because of approximations, Code Prover might not be able to keep track of

relations between variables such as x and y across several lines of code leading to an orange division by zero check. An increase in precision means that Code Prover is able to track such kinds of relations on more complex operations, leading to a green division by zero check.

- **Red, green or orange to grey:** An operation that was previously checked and showed one of red, green or orange colors is now proven unreachable and appears in grey.

As before, lower precision means considering more values from approximations. So, a conditional branch such as:

```
if (x >= 0)
```

that is unreachable can be considered reachable because of a negative value of x coming from approximations. With the increase in precision, this value might no longer be considered making it possible to prove that the branch is unreachable. If the branch is proven unreachable, red, green and orange checks inside the branch disappear.

For an individual check, it is easier to understand this change in direction of colors. However, these changes of colors in individual statements do not translate directly to changes in overall number of results of a certain color and type. For instance, some orange division by zero checks might turn green, but some green checks might also turn grey, leading to an overall increase or decrease in green division by zero checks. Therefore, when comparing results, instead of focusing on changes in overall numbers of checks, compare a few individual checks that changed color. In most cases, you will see that the color change is an expected result of an increase in precision.

In rare cases, you can see a decrease of precision (changes of colors in the opposite direction compared to what is described in this section). Typically, changes in Polyspace algorithms are vetted against a representative database of code samples to make sure that they do not cause significant decrease in precision or increase in analysis time.

Changes in Polyspace Bug Finder Results

For the same source code and analysis configuration, you might see a change in results because of improvements to the Polyspace Bug Finder analysis engine. In Bug Finder, a change in results falls in one of these categories: a new result appears, an existing result no longer shows up, or the same result appears on a different location in the source code.

Suppose that you see a new result in the current release. The new result might appear because of updates to a specific checker or from general updates to the analysis algorithm that affects the result locations for several checkers. To find more details, check the release notes of all releases between your prior release and the release you upgraded to.

- Look in the **Analysis results** section of the *Release Notes for Polyspace Bug Finder* for updates to a specific coding rule or defect checker:
 - Updates to existing defect checkers appear in a specific entry **Updated Bug Finder defect checkers**.
 - Updates to existing external coding standard checkers appear in a specific entry **Changes to external coding standards checking**.
- Look in the **Reviewing results** section of the *Release Notes for Polyspace Bug Finder* for changes in location of checker results. For instance, if a result previously appeared in separate instances of a macro, it might now be rolled up to the macro definition.

See Also

Related Examples

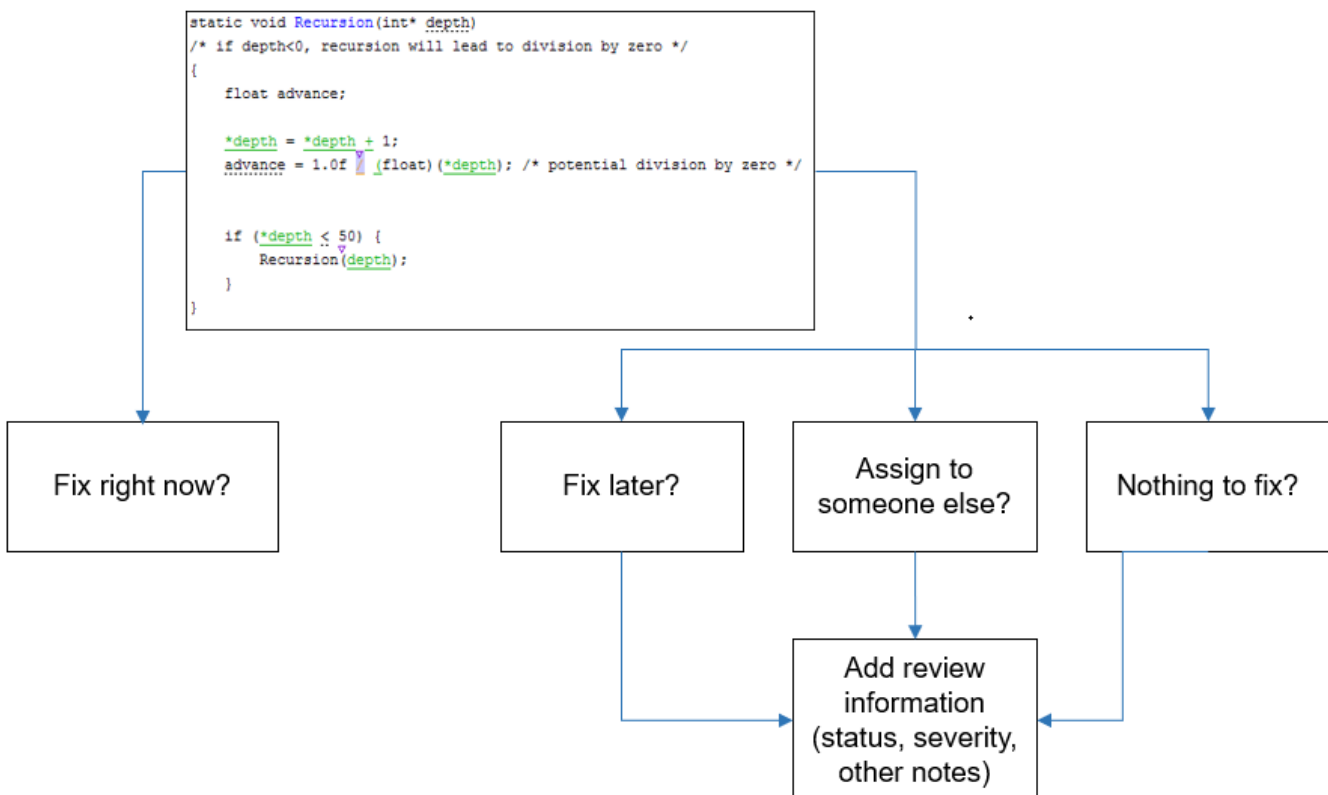
- “Migrate Polyspace Projects After Product Upgrade”

Fix or Comment Polyspace Results

Address Results in Polyspace User Interface Through Bug Fixes or Justifications

This topic describes how to add review information to Polyspace results in the user interface of the Polyspace desktop products. For a similar workflow in the Polyspace Access web interface, see “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 26-2.

Once you understand the root cause of a Polyspace finding, you can fix your code. Otherwise, add review information to your Polyspace results to fix the code later or to justify the result. You can use the information to keep track of your review progress.



If you add review information to your results file, they carry over to the results of the next analysis on the same project. If you add the same information as comments to your code (annotate), they carry over to any subsequent analysis of the code, whether in the same project or not. You can also hide results using code annotations.

Add Review Information to Results File

You can add review information either on the **Results List** or **Result Details** pane. Select a result, then set the **Severity** and **Status** fields, and optionally, enter notes with more explanations.

The screenshot shows the 'Result Details' window in Polyspace. At the top, there is a 'Variable trace' checkbox. Below it is the 'Result Review' section, which includes 'Severity' set to 'High' and 'Status' set to 'Fix'. A text box contains the comment 'Adding missing else condition.'. Below this is a yellow warning banner for a 'Non-initialized pointer' (Impact: High) with a question mark icon. The banner text reads: 'Local pointer 'pi' may be read before being initialized.'. Below the banner is a table with the following data:

	Event	File	Scope	Line
1	Declaration of variable 'pi'	dataflow.c	bug_noninitptr()	152
2	Not entering if statement (if-condition false)	dataflow.c	bug_noninitptr()	154
3	Non-initialized pointer	dataflow.c	bug_noninitptr()	159

The status indicates your response to the Polyspace result. If you do not plan to fix your code in response to a result, assign one of the following statuses:

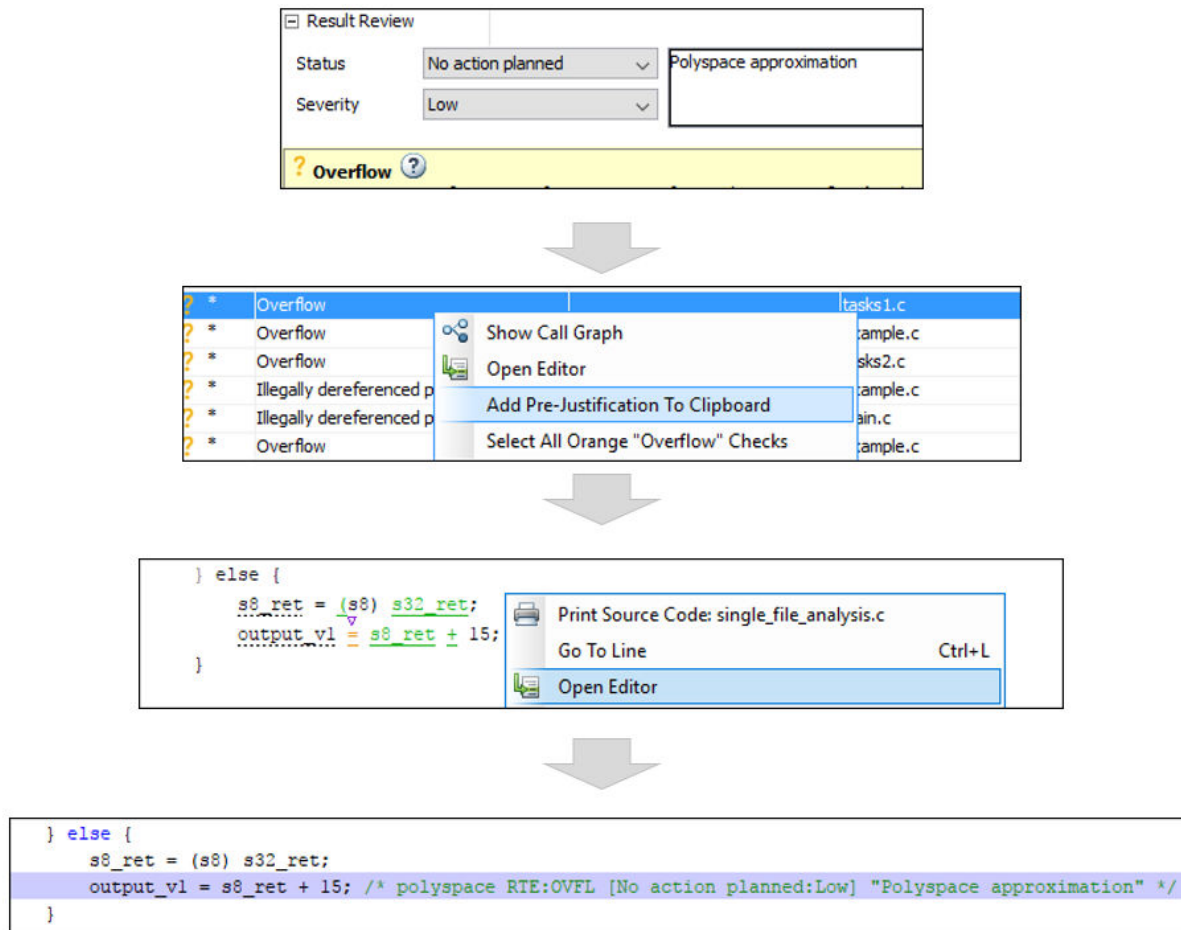
- Justified
- No Action Planned
- Not a Defect

These statuses indicate that you have given due consideration and justified that result (retained the code despite the result). Note that subsequent analyses continue to show justified results as before. For instance, a Code Prover result that was previously orange does not turn green after justification. However, during review, you can filter out justified results in one click and focus only on results that are not justified. See “Filter and Group Results in Polyspace Desktop User Interface” on page 23-2.

You can also create your own statuses to assign. See “Create Custom Review Status” on page 2-24.

Comment or Annotate in Code

You can also add specific code comments or annotations in response to Polyspace results. If you enter code comments or annotations in a specific syntax, on the next analysis of the code, the software can read them and populate the **Severity**, **Status**, and **Comment** fields in the result details.



You can either type the annotation directly or copy it from the user interface:

- In the user interface, to copy annotations, right-click a result and select **Add Pre-Justification To Clipboard**. Open your source code in an editor and paste *on the same line as* the result.
- Type the annotation *on the same line as* the result. See the annotation syntax in “Annotate Code and Hide Known or Acceptable Results” on page 30-2.

If you copy or type the annotation without explicitly assigning a status, Polyspace assumes that you have set a status of **No Action Planned**. The software hides the result from all places (except reports needed for certification³). The only exceptions are the safety-critical Code Prover run-time checks, which are hidden from the results list but not the source code. If you want to explicitly set a status, first fill the **Status** field for a result and then copy the annotation to your code. Paste on the line containing the result.

To unhide the hidden results, from the **Showing** menu, clear the box **Hide results justified in code**.

³ Reports generated from Polyspace results are typically meant for archiving and certification. Therefore, the reports contain all Polyspace results, justified or otherwise. Justified results show the justification status, for instance, **No Action Planned**, along with comments supporting the justification. These reports allow standards committees such as certification authorities to verify if a Polyspace result was justified for approved reasons.

Showing 2,699/2,699 ▼

Review Scope: All results
New results only: Off

Showing 2,699 out of 2,699 possible results
Filtered results: 0
Hidden results: 0

Hide results justified in code

Columns with active filters:
No filtered columns

Clear active filters

See Also

More About

- “Annotate Code and Hide Known or Acceptable Results” on page 30-2
- “Import Review Information from Previous Polyspace Analysis” on page 20-2

Manage Results

Filter and Group Results in Polyspace Desktop User Interface

This topic describes how to filter, group, and otherwise manage results in the user interface of the Polyspace desktop products. For a similar workflow in the Polyspace Access web interface, see “Filter and Sort Results in Polyspace Access Web Interface” on page 27-8.

When you open the results of a Polyspace analysis, you see a flat list of defects (Bug Finder), run-time checks (Code Prover), coding rule violations or other results. To organize your review, you can narrow down the list or group results by file or result type.

Family	Group	Check	File	Function
● *	Static memory	Out of bounds array index	single_file_analysis.c	reset_temperature()
● *	Static memory	Illegally dereferenced pointer	example.c	Pointer_Arithmetic()
● *	Control flow	Non-terminating call	example.c	Recursion_caller()
● *	Control flow	Non-terminating loop	main.c	interpolation()
● *	Other	Invalid use of standard library routine	example.c	Square_Root()
✕ *	Data flow	Unreachable code	initialisations.c	compute_new_coordonates()
✕ *	Data flow	Unreachable code	example.c	Pointer_Arithmetic()
✕ *	Data flow	Unreachable code	example.c	Unreachable_Code()
✕ *	Data flow	Unreachable code	single_file_analysis.c	generic_validation()
✕ *	Data flow	Unreachable code	single_file_analysis.c	generic_validation()
✕ *	Data flow	Unreachable code	main.c	interpolation()
✕ ⓘ *	Not shared	Unused variable	initialisations.c	_init_globals()
⚠ *	Static memory	Out of bounds array index	single_file_analysis.c	generic_validation()
⚠ *	Numerical	Division by zero	example.c	Recursion()
⚠ *	Data flow	Non-initialized local variable	example.c	get_oil_pressure()

Filter results

Family	Group	Check
● *	Static memory	Out of bounds array index
● *	Static memory	Illegally dereferenced pointer
● *	Other	Invalid use of standard library routine
● *	Control flow	Non-terminating call
● *	Control flow	Non-terminating loop

Group results

Family	Check
-example.c	
+ -Close_To_Zero()	
+ -File Scope	
+ -get_oil_pressure()	
+ -Non_Infinite_Loop()	
- -Pointer_Arithmetic()	
... ● *	Illegally dereferenced pointer

Some of the ways you can use filtering are:

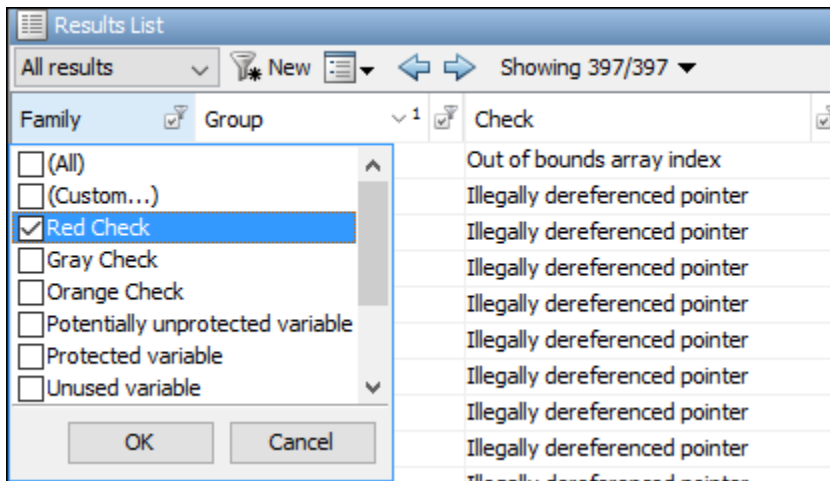
- You can display certain types of defects or run-time checks only.

For instance, in Bug Finder, you can display only high-impact defects. See “Classification of Defects by Impact” on page 18-49.

- You can display only new results found since the last analysis.
- You can display only the results that are not justified.

Filter Results

Filter Using Results List



You can filter using the columns on the **Results List** pane. Click the  icon on the column headers to see the available filters. For instance:

- To see only Bug Finder defects with high impact, from the filters on the **Information** column, clear all check boxes except **Impact: High**.
- To see only results that are not yet justified, clear the **True** filter on the **Justification** column. This column might not be visible by default. To see the column, right-click any column header and select **Justification**.

For information on justification, see “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 22-2.

For information on the columns, see:

- “Results List in Polyspace Desktop User Interface” on page 21-15
- “Results List in Polyspace Desktop User Interface” (Polyspace Code Prover)

Results found since the last analysis appear with an asterisk (*) next to them. To see only these results since the last analysis, click the **New** button. Note that if you run an analysis at the command line (or even when you run an analysis in the user interface for the first time), you have to first import from a previous analysis to create a baseline for the **New** button. See “Import Review Information from Previous Polyspace Analysis” on page 20-2.

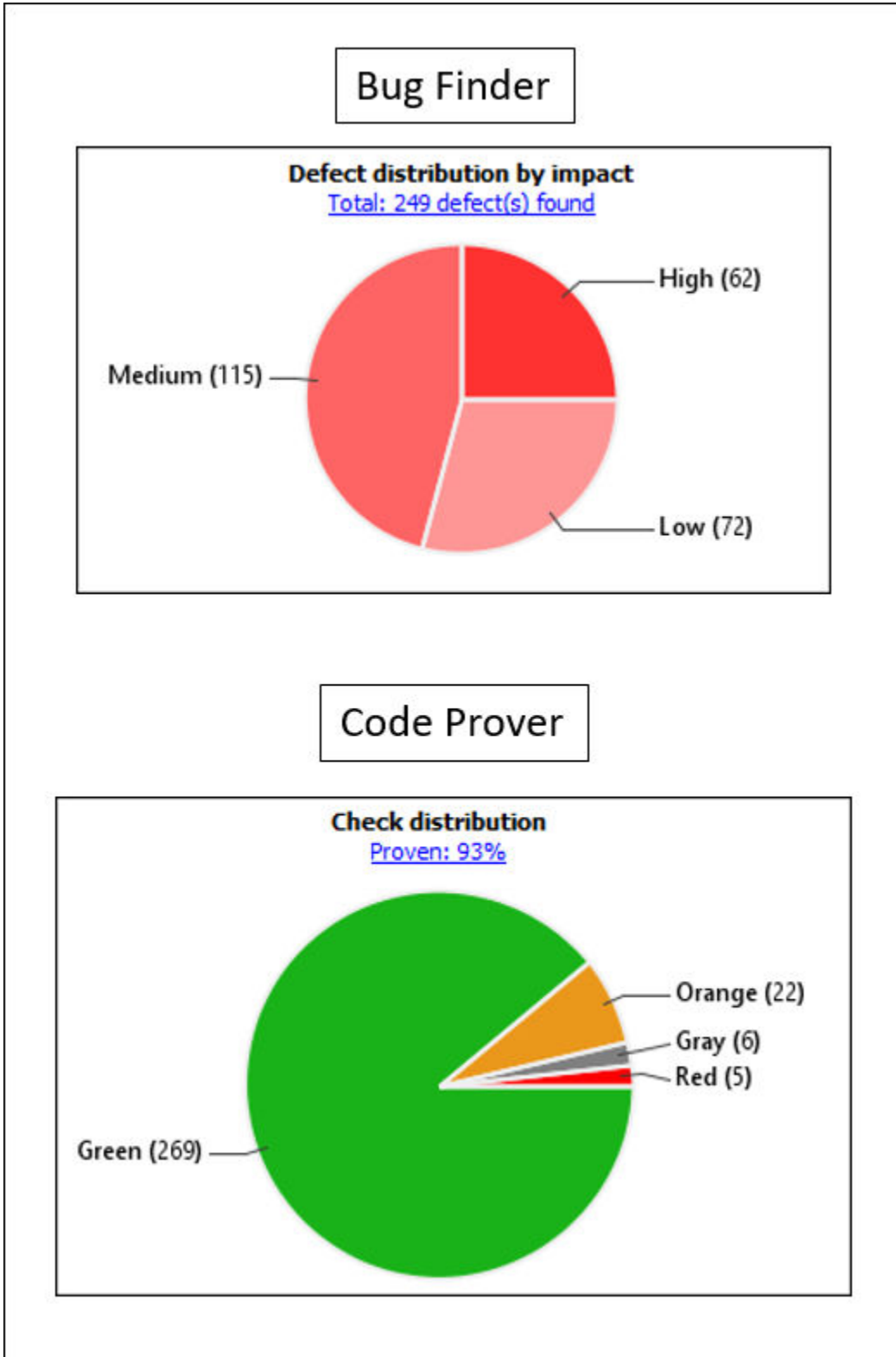
If you do not want to filter by the exact contents of a column, you can use a custom filter instead. For instance, you want to filter out subfolders of a specific folder. Instead of filtering out each subfolder in

the **Folder** column, select **Custom** from the filter dropdown. Specify the root folder name for the doesn't contain filter.

You can use wildcard characters for the custom filter. The wildcard ? represents 0 or 1 character and * represents 0 or more characters.

If you apply filters in this way, they carry over to the next analysis. You can also name and save a subset of filters for use in multiple projects. To apply the named set of filters, pick this filter set from the **All results** list. To create a new entry in this list, select **Tools > Preferences** and create your own set of filters on the **Review Scope** tab.

Filter Using Dashboard



You can click graphs on the **Dashboard** pane to filter results. For instance:

- To see only high-impact defects in Bug Finder, click the corresponding section of the **Defect distribution by impact** chart.
- To see only red checks in Code Prover, click the corresponding section of the **Check distribution** chart.

To see all results again, click the link **View all results in this scope**.

Filter Using Orange Sources

An orange source can cause multiple orange checks in Code Prover. You can display all orange checks from the same source and review them together.

For instance, in this code, the unknown value `input` can cause an overflow and a division by zero. The variable `input` is an orange source that causes two orange checks.

```
void func (int input) {
int val1;
double val2;
val1 = input++;
val2 = 1.0/input;
}
```

To begin, select **Window > Show/Hide View > Orange Sources**. You see the list of orange sources. Select an orange source to see all orange checks coming from this source.

Source Type	Name	File	Line	Max Oranges
stubbed function	get_bus_status()		-1	1
stubbed function	random_float()		-1	3
stubbed function	random_int()		-1	1
local volatile variable	get_oil_pressure.vol_i	example.c	27	2
local volatile variable	all_values_s32.tmps32	single_file_analysis.c	29	2

See Filters Used

Review Scope: Checks & Rules
New results only: On

Showing 260 out of 397 possible results
Filtered results: 137
Hidden results: 0

Hide results justified in code

Columns with active filters:
Check

On the **Results List** header, you see the number of results displayed in the format **Showing x/y**, for instance **Showing 100/250**. Click the dropdown beside this number to see the filters that are

currently active. You can also clear the active filters from this dropdown (all except the named set of filters that you picked from the **All results** dropdown).


You see this information about the filters:

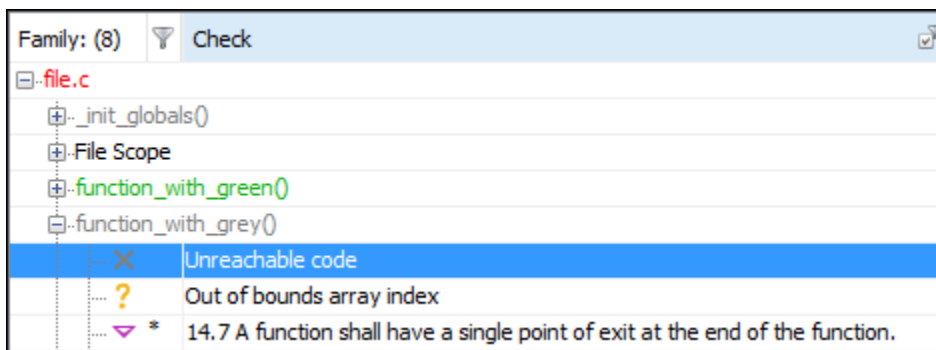
- **Review Scope:** If you pick a named set of filters from the **All results** dropdown, you see this filter set.
- **New results only:** If you use the **New** button to see only new results, you see this filter enabled.
- **Filtered results:** You see the number of results filtered in the Polyspace user interface (by any means: results list, dashboard or orange sources).
- **Hidden results:** You see the number of results hidden using code annotations. To unhide these results, clear **Hide results justified in code**.

For information on hiding results through code annotations, see “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 22-2.

- **Columns with active filters:** You see the columns in the **Results List** pane (or columns corresponding to graphs in the **Dashboard** pane) that you used to filter results.

Group Results

On the **Results List** pane, from the  list, select an option, for instance, grouping by file. Alternatively, you can click a column header to sort the column contents alphabetically.



The available options for grouping are:

- **None:** Shows results without grouping.
- **Family:** Shows results grouped by result type.

The results are organized by type: checks (Code Prover), defects (Bug Finder), global variables (Code Prover), coding rule violations, code metrics. Within each type, they are grouped further.

- The defects (Bug Finder) are organized by the defect groups. For more information on the groups, see “Defects”.
- The checks (Code Prover) are grouped by color. Within each color, the checks are organized by check group. For more information on the groups, see “Run-Time Checks” (Polyspace Code Prover).
- The global variables (Code Prover) are grouped by their usage. For more information, see “Global Variables” (Polyspace Code Prover).

- The coding rule violations are grouped by type of coding rule. For more information, see “Coding Standards”.
- The code metrics are grouped by scope of metric. For more information, see “Code Metrics”.
- **File:** Show results grouped by file.

Within each file, the results are grouped by function. The results that are not associated with a particular function are grouped under **File Scope**.

In Code Prover, the file or function name shows the worst check color in the file or function. The severity of a check color decreases in the order: red, gray, orange, green.

- **Class** (for C++ code only): Shows results grouped by class.

Within each class, the results are grouped by method. The results that are not associated with a particular class are grouped under **Global Scope**.

See Also

More About

- “Classification of Defects by Impact” on page 18-49

Generate Reports from Polyspace Results

- “Generate Reports from Polyspace Results” on page 24-2
- “Export Polyspace Analysis Results” on page 24-5
- “Export Polyspace Analysis Results to Excel by Using MATLAB Scripts” on page 24-9
- “Visualize Bug Finder Analysis Results in MATLAB” on page 24-11
- “Customize Existing Bug Finder Report Template” on page 24-15
- “Generate Report Containing MISRA C:2012 Violations, Code Metrics, and Runtime Check Results” on page 24-20

Generate Reports from Polyspace Results

This topic shows how to generate reports from results generated with a Polyspace desktop product. To generate reports from results uploaded to the Polyspace Access web server, see `polyspace-report-generator`.

To generate reports from Polyspace results, you can do one of the following:

- Run a Polyspace analysis and create a report from the analysis results. See the workflow described here.
- Specify that a report will be automatically generated after analysis. For more information on the options, see “Reporting”. Report generation immediately after analysis is supported for both desktop and server products.
- Export your results to a text file and generate graphs and statistics. See “Export Polyspace Analysis Results” on page 24-5.

Depending on the template you use, the report contains information about certain types of results from the **Results List** pane. You can see the following information about a result:

- ID: Unique number for a result for the current analysis

To identify the result in your source code, you can use the ID in the **Results List** pane of the Polyspace user interface or in your IDE if you are using a Polyspace plugin.

- Check: Defect names, MISRA C:2012 coding rule number, and so on.
- File and function
- Status, Severity, Comment: Information that you enter about a result.⁴

In Bug Finder, the report does not contain the line or column number for a result. Use the report for archiving, gathering statistics and checking whether results have been reviewed and addressed (for certification purposes or otherwise). To review a result in your source code, use the Polyspace user interface or your IDE if you are using a Polyspace plugin.

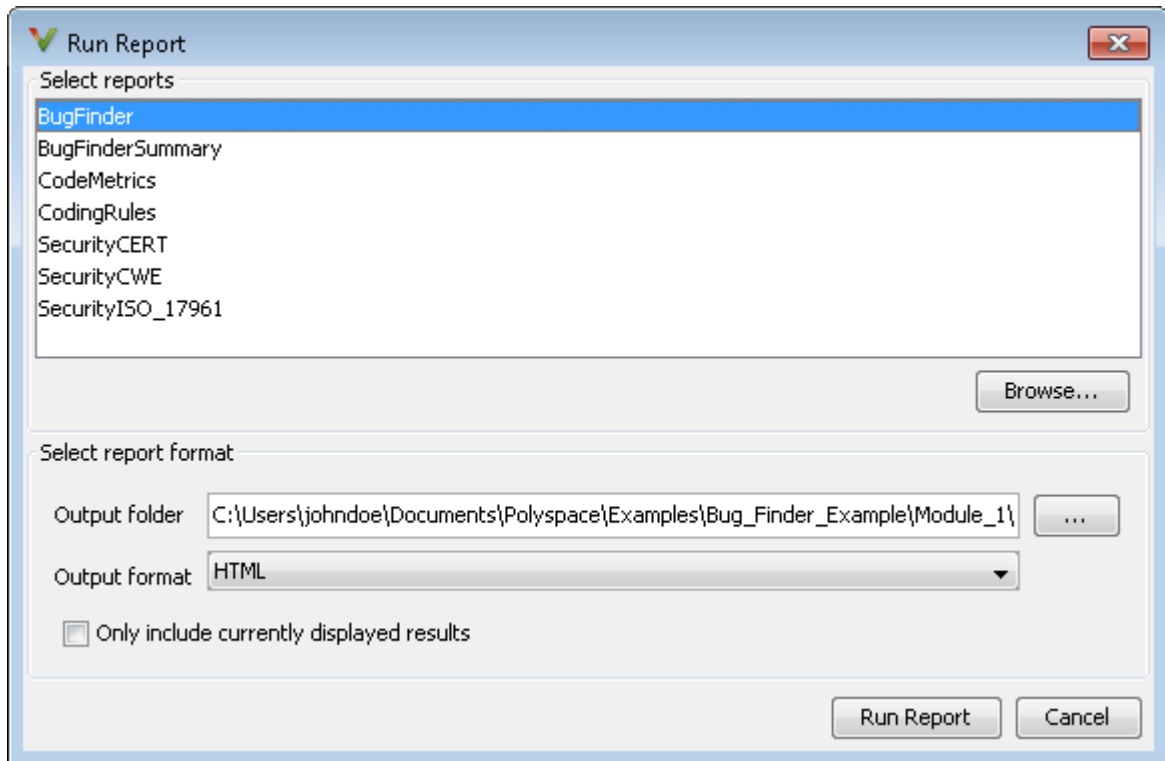
Generate Reports from User Interface

You can generate a report from your analysis results. Using a customizable template, the report presents your results in a concise manner for managerial review or other purposes.

- 1 Open your results file.
- 2 Select **Reporting > Run Report**.

The Run Report dialog box opens.

⁴ Reports generated from Polyspace results are typically meant for archiving and certification. Therefore, the reports contain all Polyspace results, justified or otherwise. Justified results show the justification status, for instance, **No Action Planned**, along with comments supporting the justification. These reports allow standards committees such as certification authorities to verify if a Polyspace result was justified for approved reasons.



- 3 Select the following options:
 - In the **Select Reports** section, select the types of reports that you want to generate. Press the **Ctrl** key to select multiple types. For example, you can select **BugFinder** and **CodeMetrics**.
 - Select the **Output folder** in which to save the report.
 - Select an **Output format** for the report.
 - If the display language (Windows) or locale (Linux) of your operating system is set to another language, you see an option to generate English reports. Select this option if you want an English report, otherwise the report is in another language.
 - If you want to filter results from your report, use filters on the **Results List** pane to display only the results that you want to report. Then, when generating reports, select **Only include currently displayed results**. You cannot display filtered reports for results downloaded from Polyspace Metrics.

For more information on filtering, see “Filter and Group Results in Polyspace Desktop User Interface” on page 23-2.

- 4 Click **Run Report**.

The software creates the specified report and opens it.

Generate Reports from Command Line

You can script the generation of reports using the `polyspace-report-generator` command.

To generate **BugFinder** and **CodeMetrics** HTML reports for results in C:\Users\johndoe\Documents\Polyspace\Examples\Bug_Finder_Example\Module_1\BF_Result, use the following options with the command:

```
SET template_path=^
"C:\Program Files\MATLAB\R2018a\toolbox\polyspace\psrptgen\templates\bug_finder"
SET bf_templates=^
%template_path%\BugFinder.rpt,%template_path%\CodingMetrics.rpt
SET results_dir=^
"C:\Users\johndoe\Documents\Polyspace\Examples\Bug_Finder_Example\Module_1\BF_Result"

polyspace-report-generator ^
-results-dir %results_dir% ^
-template %bf_templates ^
-format html
```

See Also

Generate report | Bug Finder and Code Prover report (-report-template) | Output format (-report-output-format)

More About

- “Customize Existing Bug Finder Report Template” on page 24-15
- “Export Polyspace Analysis Results” on page 24-5

Export Polyspace Analysis Results

You can export your analysis results to a tab separated values (TSV) text file, a MATLAB table, or to a standard JSON format. Using the exported content, you can:

- Generate graphs or statistics about your results that you cannot readily obtain from the user interface by using MATLAB or Microsoft Excel. For instance, for each Code Prover check type (**Division by zero**, **Overflow**), you can calculate how many checks are red, orange, or green.
- Integrate the analysis results with other checks you perform on your code.

Export Results to Text File

You can export results to a tab delimited text file (TSV) from the user interface or command line.

The exported text file uses the character encoding on your operating system. If special characters from your comments are not exported correctly in the text file, change the character encoding on your operating system before exporting.

Export Results from User Interface (Desktop Products Only)

- 1 Open your analysis results.
- 2 Export all results or only a subset of the results.
 - To export all results, select **Reporting > Export > Export All Results**.
 - If you want to filter results from your report, use filters on the **Results List** pane to display only the results that you want to report. Then, when exporting results, select **Reporting > Export > Export Currently Displayed Results**.

For more information on filtering, see “Filter and Group Results in Polyspace Desktop User Interface” on page 23-2.

- 3 Select a location to save the text file and click **OK**.

Note If you apply a review scope that sets thresholds for code metrics and you export all results, the generated file lists the results for the code metrics as **Green** (pass) or **Red** (fail) in the color column (third column). If you do not set thresholds for code metrics, the exported file shows **Not Applicable** for code metrics results in the color column.

Export Results From Command Line

Use the option `-format csv` with the `polyspace-results-export` command. For example, to generate a csv file from results file `C:\Polyspace_Workspace\myProject\Module_1\BF_Result\ps_results.psbf`, run this command:

```
polyspace-results-export -format csv -results-dir C:\Polyspace_Workspace\myProject\Module_1\BF_Result
```

Export Results to MATLAB Table

If you write MATLAB scripts to run Polyspace, you can read your Polyspace analysis results into a MATLAB table for further processing. See:

- “Visualize Bug Finder Analysis Results in MATLAB” on page 24-11
- “Visualize Code Prover Analysis Results in MATLAB” (Polyspace Code Prover)

Export Results to JSON Format

You can export Polyspace results to a JSON object. The JSON format follows the standard notation provided by the OASIS Static Analysis Results Interchange Format (SARIF).

Use the option `-format json-sarif` with the `polyspace-results-export` command. For more information, see `polyspace-results-export`.

The JSON format contains some additional information such as the checker short name and the full message that accompanies a result. Use the JSON format if you want to use this short name or message. You can also use this format for a more standardized reporting of results. For instance, if you use several static analysis tools and want to report their results in one interface by using a single parsing algorithm, you can export all the results to the standard SARIF JSON format.

View Exported Results

The exported results include the information available on the **Results List** pane in the desktop user interface or Polyspace Access web interface (except for line and column information). See:

- “Results List in Polyspace Desktop User Interface” on page 21-15
- “Results List in Polyspace Desktop User Interface” (Polyspace Code Prover)
- “Results List in Polyspace Access Web Interface” on page 25-19

Note that some **Results List** column headers might be labeled differently in the exported results file.

Some other differences in presentation between the **Results List** pane and exported results are listed below.

- The TSV file and MATLAB table contain these additional columns compared to the **Results List**:
 - **New** column — Shows whether a result is new compared to the previous run.
 - **Key** column — The entry in this column is based on the result family, result acronym, and the location of the result in the file. See also “Compare Merged Results Using Exported Keys” on page 24-6.
 - **URL** column (Polyspace Access results only) — Click the URL to open the corresponding result in the Polyspace Access interface.
- The JSON file stores the results as objects contained in a `results` array. Each object has a list of key-value pairs that store the results information, including whether the result is new compared to the last run (“`baselineState`”) and the result key. For more on the JSON format see SARIF-v2.1.0.

You cannot identify the location of a Bug Finder result in your source code via the text file. However, you can still parse the file and generate graphs or statistics about your results.

Compare Merged Results Using Exported Keys

When you merge exported analysis results from multiple modules that contain common files, you can use the **Key** and other fields from the exported results to eliminate duplicates. For instance, if you

run coding-rule checking on two different modules and merge the results, coding rule violations in common header files appear twice in the results.

To eliminate duplicates, compare the **Key** and **File** values of the results. If two results have the same **Key** and **File** values, one is a duplicate of the other.

By default, each result key is based on the result family (for instance `numerical`), result short name (for instance `float_ovfl`), and the location of the result in the file. To generate more localized keys for results that are located inside a function body, use the `-key-mode function-scope` option with the `polyspace-results-export` or the `polyspace-report-generator` commands. The commands generate keys that are based on the result family, result short name, and the location of the result within the function body. You can then identify duplicates more accurately by also comparing the **Function** values.

In rare instances, results that have the same key and the same location inside a file or function body might not be duplicates. In those instances you need to compare the results manually, for instance by comparing the result messages in the user interface, to determine whether the results are duplicates or not.

Enable Function Scope for Exported Keys

To enable the function scope for exported keys:

- **At The Command Line**

Enter either of these commands to export locally stored results:

```
polyspace-results-export -results-dir folderPath -key-mode function-scope -format csv
```

or

```
polyspace-report-generator -generate-results-list-file -results-dir folderPath -key-mode function-scope
```

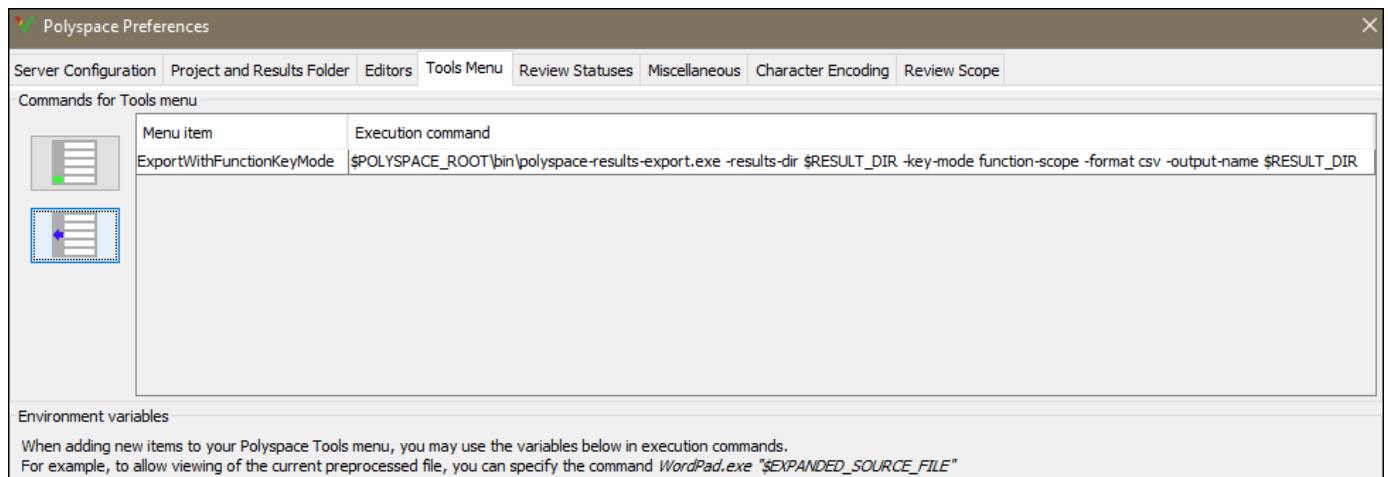
where *folderPath* is the path of the folder that contains the Polyspace analysis results.

Make sure that you run these commands from a location where you have write permissions, or use option `-output-name` to specify a location to store the generated file of exported results.

- **In the Polyspace Desktop User Interface**

Create a menu item by going to **Tools > Preferences** and entering this command on the **Tools Menu** tab:

```
$POLYSpace_ROOT\bin\polyspace-results-export.exe -results-dir $RESULT_DIR  
-key-mode function-scope -format csv -output-name $RESULT_DIR
```



You can then export results by using the menu item you created from **Tools > External Tools**.

To export results with the default key mode (without function location), use the **Reporting > Export** menu.

When you export results with the function scope enabled, the key entries for results that are inside a function have a FN prefix.

See Also

polyspace-results-export

Related Examples

- “Visualize Bug Finder Analysis Results in MATLAB” on page 24-11
- “Export Polyspace Analysis Results to Excel by Using MATLAB Scripts” on page 24-9

Export Polyspace Analysis Results to Excel by Using MATLAB Scripts

You can export the results of a Bug Finder or Code Prover analysis to an Excel report. See “Export Polyspace Analysis Results” on page 24-5. The report contains Polyspace results in a tab-delimited text file with predefined content and formatting.

You can also create Excel reports with your own content and formatting. Automate the creation of this report by using MATLAB scripts.

Report Result Summary and Details in One Worksheet

This example shows a sample script for generating Excel reports from Polyspace results.

The script adds two worksheets to an Excel workbook. The worksheets report content from the Polyspace results in `polyspaceroot\polyspace\examples\cxx\Code_Prover_Example\Module_1\CP_result`. Here, `polyspaceroot` is the Polyspace installation folder, such as `C:\Program Files\Polyspace\R2019a`.

Each worksheet contains the summary and details for a specific type of Polyspace result:

- **MISRA C:2012:** This worksheet contains a summary of MISRA C: 2012 rule violations in the Polyspace results. The summary is followed by details of each MISRA C: 2012 violation.
- **RTE:** This worksheet contains a summary of run-time errors that Code Prover found. The summary is followed by details of each run-time error.

```
% Copy a demo result set to a temporary folder.
resPath = fullfile(polyspaceroot,'polyspace','examples','cxx', ...
    'Code_Prover_Example','Module_1','CP_Result');
userResPath = tempname;
copyfile(resPath,userResPath);

% Read results into a table.
results = polyspace.CodeProverResults(userResPath);
resultsTable = results.getResults;

% Delete any existing file and create new file
filename = 'polyspace.xlsx';
if isfile(filename)
    delete(filename)
end

% Disable warnings about adding new worksheets
warning('off','MATLAB:xlswrite:AddSheet')

% Write MISRA summary to the MISRA 2012 worksheet
misraSummaryTable = results.getSummary('misraC2012');
writetable(misraSummaryTable, filename, 'Sheet', 'MISRA 2012');

% Write MISRA results to the MISRA 2012 worksheet
misraDetailsTable = resultsTable(resultsTable.Family == 'MISRA C:2012',:);
detailsStartingCell = strcat('A',num2str(height(misraSummaryTable)+ 4));
writetable(misraDetailsTable, filename, 'Sheet', 'MISRA 2012', 'Range', ...
    detailsStartingCell);

% Write runtime summary to the RTE worksheet
rteSummaryTable = results.getSummary('runtime');
writetable(rteSummaryTable, filename, 'Sheet', 'RTE');

% Write runtime results to the RTE worksheet
rteResultsTable = resultsTable(resultsTable.Family == 'Run-time Check',:);
detailsStartingCell = strcat('A',num2str(height(rteSummaryTable)+ 4));
writetable(rteResultsTable, filename, 'Sheet', 'RTE', 'Range', detailsStartingCell);
```

The key functions used in the example are:

- `polyspace.CodeProverResults`: Read Code Prover results into a table.
- `writetable`: Write a MATLAB table to a file. If the file name has the extension `.xlsx`, the function writes to an Excel file.

To specify the content to write to the Excel sheet, use these name-value pairs:

- Use the name `Sheet` paired with a sheet name to specify a worksheet in the Excel workbook.
- Use the name `Range` paired with a cell name to specify the starting cell in the worksheet where the writing begins.

Control Formatting of Excel Report

Though you can control the content exported to the Excel report by using the preceding method, the method has limited formatting options for the report.

To format the Excel report on Windows systems, access the COM server directly by using `actxserver`. For example, Technical Solution 1-QLD4K uses `actxserver` to establish a connection between MATLAB® and Excel, write data to a worksheet, and specify the colors of the cells.

See also “Get Started with COM”.

See Also

More About

- “Export Polyspace Analysis Results” on page 24-5

Visualize Bug Finder Analysis Results in MATLAB

After a Polyspace analysis, you can read your results to a MATLAB table. Using the table, you can generate graphs or statistics about your results. If you have MATLAB Report Generator, you can include these tables and graphs in a PDF or HTML report.

Export Results to MATLAB Table

To read existing Polyspace analysis results into a MATLAB table, use a `polyspace.BugFinderResults` object associated with the results.

For instance, to read the demo results in the read-only subfolder `polyspace/examples/cxx/Bug_Finder_Example/Module_1/BF_Result` of the MATLAB installation folder, copy the results to a writable folder and read them:

```
resPath = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'Module_1', 'BF_Result');

userResPath = tempname;
copyfile(resPath, userResPath);

resObj = polyspace.BugFinderResults(userResPath);
resSummary = getSummary(resObj);
resTable = getResults(resObj);
```

`resSummary` and `resTable` are two MATLAB tables containing summary and details of the Polyspace results.

Alternatively, you can run a Polyspace analysis on C/C++ source files using a `polyspace.Project` object. After analysis, the `Results` property of the object contains the results. See “Run Polyspace Analysis by Using MATLAB Scripts” on page 5-9.

Generate Graphs from Results and Include in Report

After reading the results to a MATLAB table, you can visualize them in a convenient format. If you have MATLAB Report Generator, you can create a PDF or HTML report that contains your visualizations.

This example creates a pie chart showing the distribution of showing the distribution of defects by defect groups on page 18-43, and includes the chart in a report.

```
% This example shows how to create a pie chart from your
% results and append it to a report.

%% Generate Pie Chart from Polyspace Results

% Copy a demo result set to a temporary folder.
resPath = fullfile(polyspaceroot, 'polyspace', 'examples', 'cxx', ...
    'Bug_Finder_Example', 'Module_1', 'BF_Result');
userResPath = tempname;
copyfile(resPath, userResPath);

% Read results into a table.
```

```

resObj = polyspace.BugFinderResults(userResPath);
resTable = getResults(resObj);

% Eliminate results that are not defects.
matches = (resTable.Family == 'Defect');
defectTable = resTable(matches ,:);

% Create a pie chart showing distribution of defects.
defectGroupList = removecats(defectTable.Group);
pieDefects = pie(defectGroupList);
labels = get(pieDefects(2:2:end), 'String');
set(pieDefects(2:2:end), 'String', '');
legend(labels, 'Location', 'bestoutside')

% Save the pie chart.
print('file', '-dpng');

%% Append Pie Chart to Report
% Requires MATLAB Report Generator

% Create a report object.
import mlreportgen.dom.*;
report = Document('PolyspaceReport', 'html');

% Add a heading and paragraph to the report.
append(report, Heading(1, 'Bug Finder Defect Distribution Graph'));
paragraphText = ['The following graph shows the distribution of ' ...
                 'defects in your code.'];
append(report, Paragraph(paragraphText));

% Add the image to the report.
chartObj = Image('file.png');
append(report, chartObj);

% Add another heading and paragraph to the report.
append(report, Heading(1, 'Defect Details'));
paragraphText = ['The following table shows the defects ' ...
                 'in your code.'];
append(report, Paragraph(paragraphText));

% Add the table of defects to the report.
reducedInfoTable = defectTable(:, {'File', 'Function', 'Check', ...
                                   'Status', 'Severity', 'Comment'});
reducedInfoTable = sortrows(reducedInfoTable, [1 2]);
tableObj = MATLABTable(reducedInfoTable);
tableObj.Style = {Border('solid', 'black'), ColSep('solid', 'black'), ...
                 RowSep('solid', 'black')};
append(report, tableObj);

% Close and view the report in a browser.
close(report);
rptview(report.OutputPath);

```

The key functions used in the example are:

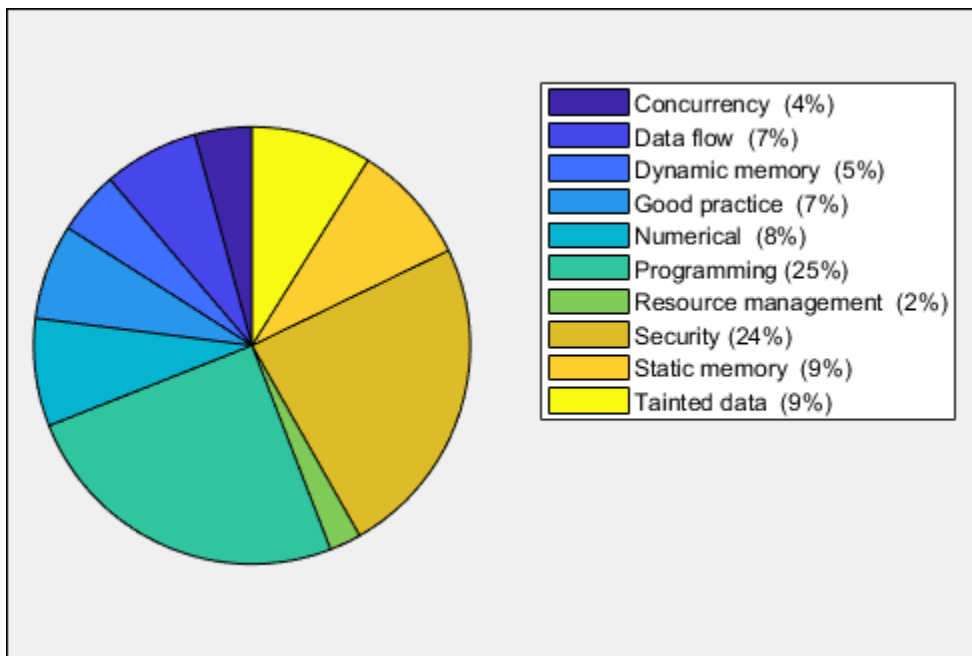
- `polyspace.BugFinderResults`: Read Bug Finder results into a table.

- `pie`: Create pie chart from a categorical array. You can alternatively use the function `histogram` or `heatmap`.

To create histograms, replace `pie` with `histogram` in the script and remove the pie chart legends.

- `mlreportgen.dom.Document`: Create a report object that specifies the report format and where to store the report.
- `append`: Append contents to the existing report.

When you execute the script, you see a distribution of defects by defect group. The script also creates an HTML report that contains the graph and table of Polyspace defects.



You can use any criteria to remove rows from the results table before reporting. The preceding example uses the criteria that the result must be from the defect family. See also Bug Finder result families.

```
matches = (resTable.Family == 'Defect');
defectTable = resTable(matches ,:);
```

Instead, you can use another criteria. For instance, you can remove results in header files and retain the results from source files only.

```
sourceExtensions = [".c", ".cpp", ".cxx"];
fileNameStrings = string(resTable.File);
matches = endsWith(fileNameStrings, sourceExtensions);
sourceTable = resTable(matches ,:);
```

See Also

Related Examples

- “Export Polyspace Analysis Results” on page 24-5
- “Export Polyspace Analysis Results to Excel by Using MATLAB Scripts” on page 24-9

Customize Existing Bug Finder Report Template

In this example, you learn how to customize an existing report template to suit your requirements. A report template allows you to generate a report from your analysis results in a specific format. If an existing report template does not suit your requirements, you can change certain aspects of the template.

For more information on the existing templates, see `Bug Finder` and `Code Prover` report (`-report-template`).

Prerequisites

Before you customize a report template:

- See whether an existing report template meets your requirements. Identify the template that produces reports in a format close to what you need. You can adapt this template.

To test a template, generate a report from sample results using the template. See “Generate Reports from Polyspace Results” on page 24-2.

- Make sure you have MATLAB Report Generator installed on your system.

In this example, you modify the **BugFinder** template that is available in Polyspace Bug Finder.

View Components of Template

A report template can be broken into components in MATLAB Report Generator. Each component represents some of the information that is included in a report generated using the template. For example, the component **Title Page** represents the information in the title page of the report.

In this example, you view the components of the **BugFinder** template.

- 1 Add paths to Polyspace-specific report components by pointing to subfolders of your Polyspace installation folder. At the MATLAB command prompt, enter:

```
addpath(fullfile(polyspaceroot, 'toolbox', 'polyspace', 'psrptgen', 'psrptgen'));
addpath(fullfile(polyspaceroot, 'toolbox', 'polyspace', 'psrptgen', 'templates'));
```

Here, *polyspaceroot* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`. If you integrate MATLAB and Polyspace, you can use the `polyspaceroot` function in MATLAB to find the installation folder location. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

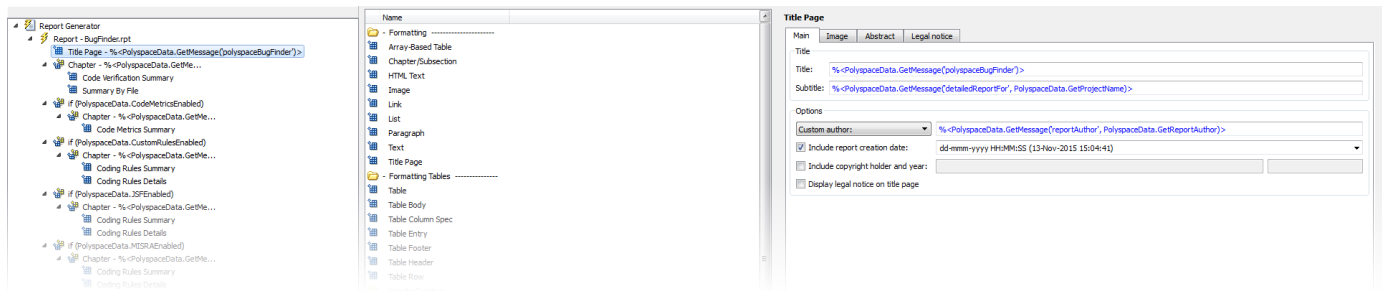
- 2 Open the Report Explorer interface. At the MATLAB command prompt, enter:

```
report
```

- 3 Open the **BugFinder** template in the Report Explorer interface.

The **BugFinder** template is in `polyspaceroot/toolbox/polyspace/psrptgen/templates/bug_finder` where *polyspaceroot* is the Polyspace installation folder.

Your template opens in the Report Explorer. On the left pane, you can see the components of the template. You can click each component and view the component properties on the right pane.



Some components of the **BugFinder** template and their purpose are described below.

Component	Purpose
Title Page (MATLAB Report Generator)	Inserts title page in the beginning of report
Chapter/Subsection (MATLAB Report Generator)	Groups portions of report into sections with titles
Code Verification Summary	Inserts summary table of Polyspace analysis results
Logical If (MATLAB Report Generator)	Executes child components only if a condition is satisfied
Run-time Checks Summary Ordered by File	Inserts a table with Polyspace Bug Finder defects grouped by file

To understand how the template works, compare the components in the template with a report generated using the template.

For more information on the components, see “Work with Components” (MATLAB Report Generator). For information on Polyspace-specific components, see “Generate Reports”.

Note Some of the component properties are set using internal expressions. Although you can view the expressions, do not change them. For instance, the conditions specified in the **Logical If** components in the **BugFinder** template are specified using internal expressions.

Change Components of Template

In the Report Explorer interface, you can:

- Change properties of existing components of your template.
- Add new components to your template or remove existing components.

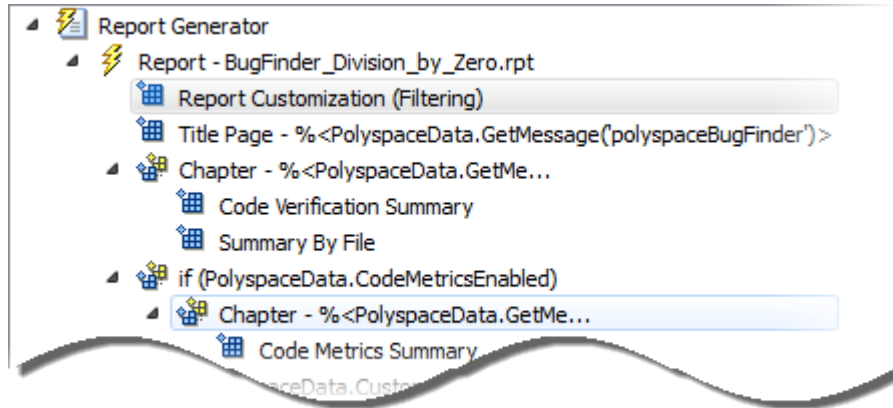
In this example, you add a component to the **BugFinder** template so that the template includes only **Integer division by zero** and **Float division by zero** defects in a report.

- 1 Open the **BugFinder** template in the Report Explorer interface and save it elsewhere with a different name, for instance, **BugFinder_Division_by_Zero**.

- 2 Add a new global component that filters every defect except division by zero from the **BugFinder_Division_by_Zero** template. The component is global because it applies to the full report and not one chapter of the report.

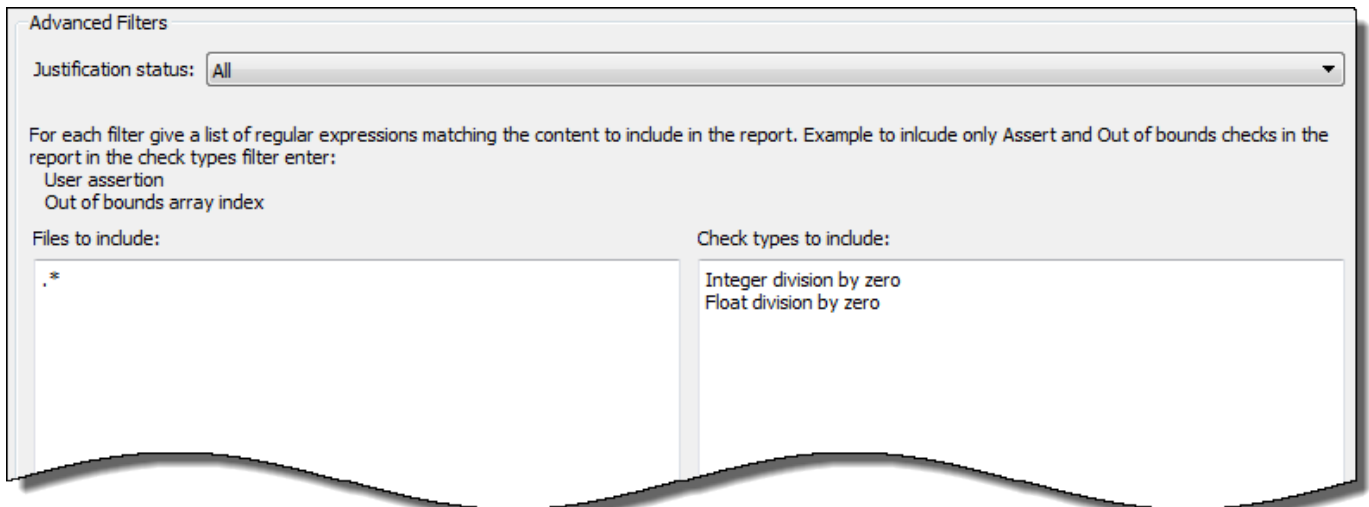
To perform this action:

- a Drag the component **Report Customization (Filtering)** located under **Polyspace** in the middle pane and place it above the **Title Page** component. The positioning of the component ensures that the filters apply to the full report and not one chapter of the report.



- b Select the **Report Customization (Filtering)** component. On the right pane, you can set the properties of this component. By default, the properties are set such that all results are included in the report.

To include only **Integer division by zero** and **Float division by zero** defects, under the **Advanced Filters** group, enter Integer division by zero and Float division by zero in the **Check types to include** field.



You can also use MATLAB regular expressions in this field to exclude results. For instance, to exclude the result **Dead code**, enter `^(?!Dead code) .*`. The report generator applies the regular expressions against the Polyspace result names. For instance:

- The caret ^ indicates that the subsequent pattern must be at the beginning of the string.
- The characters (?!pattern) .* indicates that the subsequent pattern must not appear in the string.

Together, the regular expression `^(?!Dead code) .*` indicates that Polyspace result names beginning with `Dead code` must be excluded from the report. See “Regular Expressions”.

You can toggle between activating and deactivating this component. Right-click the component and select **Activate/Deactivate Component**.

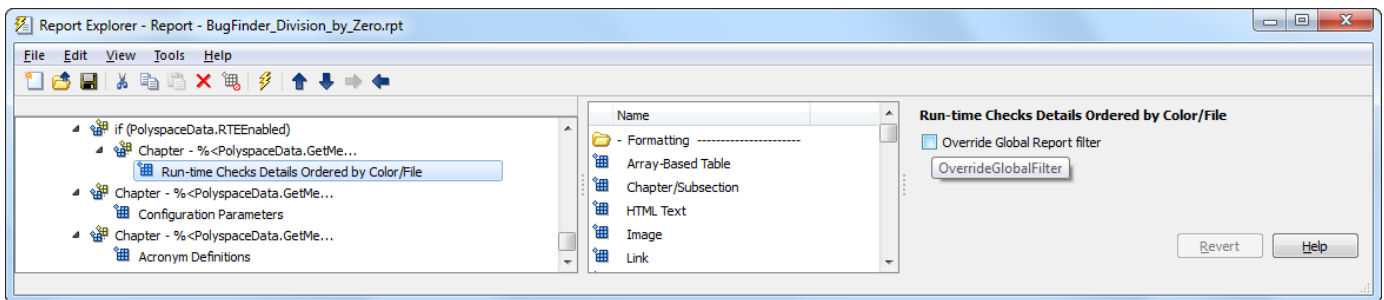
- 3 Change an existing chapter-specific component so that it does not override the global filter you applied in the previous step. If you prevent the overriding, the chapter-specific component follows the filtering specifications in the global component.

To perform this action:

- a On the left pane, select the **Run-time Checks Details Ordered by Color/File** component. This component produces tables in the report with details of run-time checks found in Polyspace Bug Finder.

The right pane shows the properties of this component.

- b Clear the **Override Global Report** filter box.



Save the **BugFinder_Division_by_Zero** template after making your changes.

- 4 In the Polyspace user interface, create a report using both the **BugFinder** and **BugFinder_Division_by_Zero** template from results containing division by zero defects. Compare the two reports.

For instance:

- a Open **Help > Examples > Bug_Finder_Example.psprj**.

The demo result contains **Integer division by zero** and **Float division by zero** defects.

- b Create a PDF report using the **BugFinder** template. See “Generate Reports from Polyspace Results” on page 24-2.

In the report, open **Chapter 5. Defects** (in your version of the product, the chapter number might be different). *You can see all defects from the example result.* Close the report.

- c Create a PDF report using the **BugFinder_Division_by_Zero** template. In the Run Report window, use the **Browse** button to add the **BugFinder_Division_by_Zero** template to the existing template list.

In the report, open **Chapter 6. Defects** (in your version of the product, the chapter number might be different). *You see only **Integer division by zero** and **Float division by zero** defects.*

Note After you add the template to the existing list of templates, before generating the report, make sure to select the newly added template.

Generate Report Containing MISRA C:2012 Violations, Code Metrics, and Runtime Check Results

To obtain a report that contains the Code Prover results, all code metrics, and MISRA C:2012 violations, run the command `polyspace-report-generator`. Generate a combined report containing these results:

- MISRA C:2012. See “MISRA C:2012 Directives and Rules”.
- Code metrics. See “Code Metrics”.
- Stack usage metrics. See Stack Usage Metrics (Polyspace Code Prover).
- Run-time checks. See “Run-Time Checks” (Polyspace Code Prover).

For more information, see `polyspace-report-generator`.

Code Prover does not check for violations of coding standards, such as MISRA C:2012. Use Bug Finder to check for coding rule violations. Code Prover computes the stack usage code metrics only. The other code metrics are calculated by Bug Finder.

Prerequisite

- Before generating the report, confirm that your installed Polyspace version is R2021b or later.
- To use the C source file used in this example, navigate to `polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources\numerical.c`. Substitute `polyspaceroot` with your Polyspace installation path, for instance, `C:\Program Files\Polyspace\R2023a`. See also “Installation Folder”.

Obtain Code Metrics and Coding Rules Results by Using Bug Finder

Run a Bug Finder analysis and store the results.

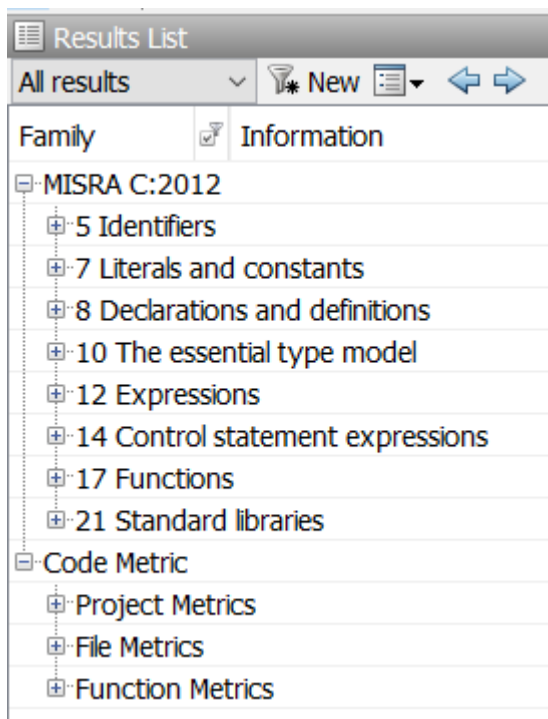
- 1 Run Bug Finder on `numerical.c` to check for coding rule standard violations, and then store the results in a folder named `BFResults`. For instance, at the command line, enter:

```
polyspace-bug-finder
-sources "polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources\numerical.c"
-results-dir BFResults -code-metrics -misra3 all-rules -lang c -checkers none
```

The command runs Bug Finder on `numerical.c` and stores the results in a folder named `BFResults`. Bug Finder checks for violations of MISRA C:2012 coding rules and computes the code metrics.

In the next steps, Code Prover performs an exhaustive check for run-time errors and other issues. In such a case, activating the Bug Finder defects might be redundant. Polyspace Report Generator does not support putting Code Prover run-time errors and Bug Finder defects in the same report. The defect checkers are deactivated in this step.

- 2 Navigate to `Current Folder\BFResults` and open the file `ps_results.psbf`. Verify that the results contain MISRA C;2012 violations and code metrics results.



Obtain Run Time Check and Stack Usage Results by Using Code Prover

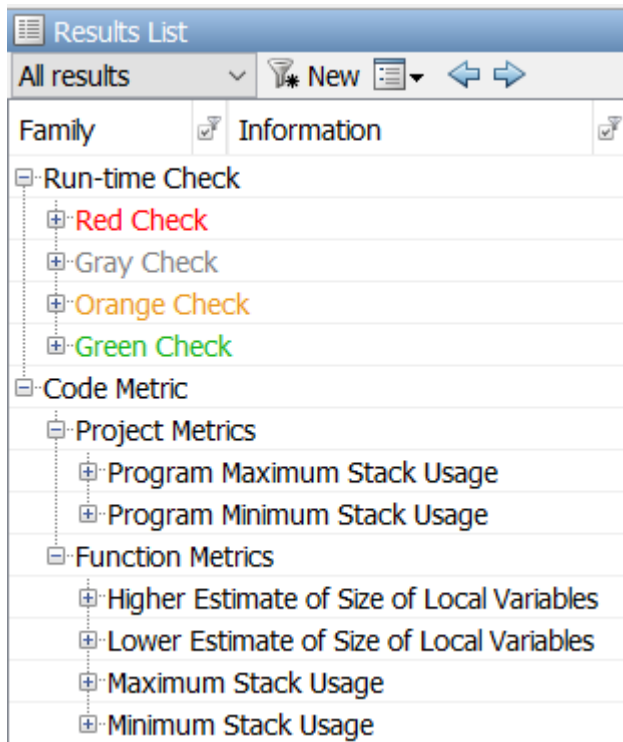
Run a Code Prover analysis and store the results.

- 1 Run Code Prover on `numerical.c` to check for run-time issues and to calculate stack usage metrics. Store the results in a folder named `CPResults`. For instance, at the command line, enter:

```
polyspace-code-prover
-sources "polyspace\examples\cxx\Bug_Finder_Example\sources\numerical.c"
-results-dir CPResults -stack-usage -lang c -main-generator
```

The command runs Code Prover on `numerical.c` and stores the results in a folder named `CPResults`. Because `numerical.c` does not have a `main()` function, specify the option `Verify module or library (-main-generator)`.

- 2 Navigate to *Current Folder*\`CPResults` and open the file `ps_results.pscp`. Verify that the results contain run-time checks and stack usage results.



Generate a Combined Report

After generating the Bug Finder and Code Prover results, summarize the results into a single report. To generate the report, specify an appropriate template and the results folders as inputs to the command `polyspace-report-generator`.









- 1 Before executing the report generation command, select an appropriate template. For this example, the report must contain a chapter each for code metrics, Code Prover runtime checks, and MISRA C:2012 violations. The template `Developer.rpt` accommodates all of these results.

To include additional chapters about coding standards such as AUTOSAR C++14 or CERT C/C++, modify the existing templates. See “Customize Existing Bug Finder Report Template” on page 24-15.

- 2 Start the report generation by using the command `polyspace-report-generator`. Specify the template `Developer.rpt` as the input to `-template`. Provide the folders containing the Bug Finder and Code Prover results as inputs to `-results-dir`. At the command line, enter:

```
polyspace-report-generator
-template "polyspaceroot\toolbox\polyspace\psrptgen\templates\Developer.rpt"
-results-dir "CPRResults","BFResults" -output-name combined_report -format PDF
```

The file `combined_report.pdf` is saved in your current folder. Open the file and verify that the report contains code metrics, Code Prover run-time checks, and MISRA C:2012 violations.

-  Chapter 1. Polyspace Code Verification Summary
-  Chapter 2. Polyspace Run-Time Checks Statistics
-  Chapter 3. Code Metrics
-  Chapter 4. MISRA C:2012 Guidelines
-  Chapter 5. Polyspace Run-Time Checks Results
-  Chapter 6. Global Variables
-  Chapter 7. Appendix 1 - Configuration Settings
-  Chapter 8. Appendix 2 - Definitions

See Also

[polyspace-report-generator](#) | [polyspace-bug-finder](#)

Related Examples

- “Customize Existing Bug Finder Report Template” on page 24-15
- “Export Polyspace Analysis Results” on page 24-5
- “Generate Reports from Polyspace Results” on page 24-2
- “Migrate Code Prover Workflows for Checking Coding Standards and Code Metrics to Bug Finder” on page 16-55

Review Results on Web Browser

Interpret Polyspace Bug Finder Results

- “Interpret Bug Finder Results in Polyspace Access Web Interface” on page 25-2
- “Investigate the Cause of Empty Results List” on page 25-7
- “Dashboard in Polyspace Access Web Interface” on page 25-9
- “Code Metrics Dashboard in Polyspace Access Web Interface” on page 25-11
- “Quality Objectives Dashboard in Polyspace Access” on page 25-14
- “Results List in Polyspace Access Web Interface” on page 25-19
- “Source Code in Polyspace Access Web Interface” on page 25-21
- “Result Details in Polyspace Access Web Interface” on page 25-26
- “Call Hierarchy in Polyspace Access Web Interface” on page 25-28
- “Configuration Settings in Polyspace Access Web Interface” on page 25-30
- “Review History in Polyspace Access Web Interface” on page 25-33
- “Create Bug Tracking Tool Tickets from the Polyspace Access Web Interface” on page 25-35

Interpret Bug Finder Results in Polyspace Access Web Interface

This topic shows how to review Bug Finder results in the Polyspace Access web interface. For a similar workflow in the user interface of the Polyspace desktop products, see “Interpret Bug Finder Results in Polyspace Desktop User Interface” on page 21-2. To see how to review results of Polyspace as You Code in IDEs, see “Review Polyspace as You Code Results in IDEs”.

When you open the results of a Bug Finder analysis in the **REVIEW** view of Polyspace Access, you see a list on the **Results List** pane. The results consist of defects, coding rule violations or code metrics.

You can first narrow down the focus of your review:

- Use filters in the toolstrip to narrow down the list. For instance, you can focus on the high-impact defects.
- Click the a column header in the **Results List** to sort the list according to the content of that column. For instance you can sort by **Group** or by **File**.

Once you narrow down and sort the list, you can begin reviewing individual results. This topic describes how to review a result.

The screenshot shows the Polyspace Access web interface. The top navigation bar includes 'Dashboard', 'Running Checks', 'Defects', 'Coding Standards', 'Code Metrics', and 'Global Variables'. Below this is a toolbar with 'To Do', 'In Progress', and 'Done' buttons, along with a 'Show only' dropdown set to 'Comment, filename, etc.' and a 'Filter out' dropdown also set to 'Comment, filename, etc.'. The main content area is divided into two panes: 'Results List' and 'Result Details'.

The 'Results List' pane displays a table of defects. The table has columns for 'Family', 'ID', 'Type', 'Group', and 'Check'. The row for ID 77758 is selected and highlighted in blue. This row shows a 'Defects' type, 'Dynamic memory' group, and the check 'Invalid free of pointer'.

The 'Result Details' pane shows the details for the selected result. It includes a 'Status' dropdown set to 'Unreviewed', a 'Severity' dropdown set to 'Unset', and an 'Assigned to' field. Below this is a yellow warning box with the text: 'Invalid free of pointer (Impact: High) Freed pointer does not come from an adapted allocation.' Underneath is a table with columns 'Event', 'File', and 'Scope'. The table has three rows, with the third row selected and highlighted in blue. This row shows 'Invalid free of pointer' in the 'Event' column, 'dynamicmemory.c' in the 'File' column, and 'bug_badfree()' in the 'Scope' column.

The 'Source Code' pane at the bottom shows the source code for 'dynamicmemory.c'. The code includes a function definition for 'bug_badfree(int i)'. The line 'free(p);' is highlighted in red, and a comment next to it reads: '/* Defect: free on a non-allocated memory */'. A blue arrow points from the 'See source code.' annotation to this line.

Annotations above the interface indicate the workflow: 'Select a result.' points to the selected row in the 'Results List' table; 'Read result explanation.' points to the 'Result Details' pane; and 'See source code.' points to the source code in the 'Source Code' pane.

To begin your review, select a result in the list.



Interpret Result Details Message

The screenshot displays the Polyspace Bug Finder interface. On the left, the 'Result Details' pane shows a message: 'Invalid use of standard library integer routine (Impact: High)'. Below the message is a table with columns 'Event', 'File', and 'Scope'. The 'Source Code' pane shows the corresponding C code. On the right, the 'Contextual Help' pane provides a detailed description of the error, a list of examples (like 'Absolute Value of Large Negative'), and check information (Group: Numerical, Language: C/C++, Default: On, Command-Line Syntax: INT_57D_118, Impact: High, CWE ID: 227, 369, 682, 872). Callouts with arrows point to specific elements: 'Open contextual help.' points to a question mark icon; 'Read brief explanation.' points to the message text; 'Read detailed explanation with examples.' points to the description and examples sections; 'Check external standards.' points to the CWE ID at the bottom.

Interpret Message

The first step is to understand what is wrong. Read the message on the **Result Details** pane and the related line of code on the **Source Code** pane.

Seek Additional Resources for Help

Sometimes, you need additional help for certain results. Click the  icon to open a help page for the selected result. See code examples illustrating the result. Check external standards such as CERT-C that provide additional rationale for fixing the issue. When available, click the  icon to see fix suggestions for the defect.

At this point, you might be ready to decide whether to fix the issue or not. Once you identify a fix, it might help to review all results of that type together.

Find Root Cause of Result

Sometimes, the root cause might be far from the actual location where the result is displayed. For instance, a variable that you read might be non-initialized because the initialization is not reachable. The defect is shown when you read the variable, but the root cause is perhaps a previous `if` or `while` condition that is always false.

Navigate to Related Events

Typically, the **Result Details** pane shows one sequence of events that leads to the result. The **Source Code** pane also highlights these events.

The screenshot shows the 'Result Details' window for a bug in 'dataflow.c / bug_noninitvar()'. The main area contains a yellow warning box for a 'Non-initialized variable' (Impact: High) with the following text:

Local variable *value* is read before it is initialized.
Additional Info:
Risk: Reading non-initialized memory can result in unexpected values.
Fix: Initialize the local variable before use.

Below the warning box is a table of event tracebacks:

	Event	File	Score
1	Declaration of variable '...	dataflow.c	bug_
2	Not entering if statemen...	dataflow.c	bug_
3	Non-initialized variable	dataflow.c	bug_

The right-hand sidebar contains the following fields:

- Status:** To investigate
- Severity:** Unset
- Assigned to:** Type username or ...
- Ticket:** + link icon
- Comment:** Enter your comment here...

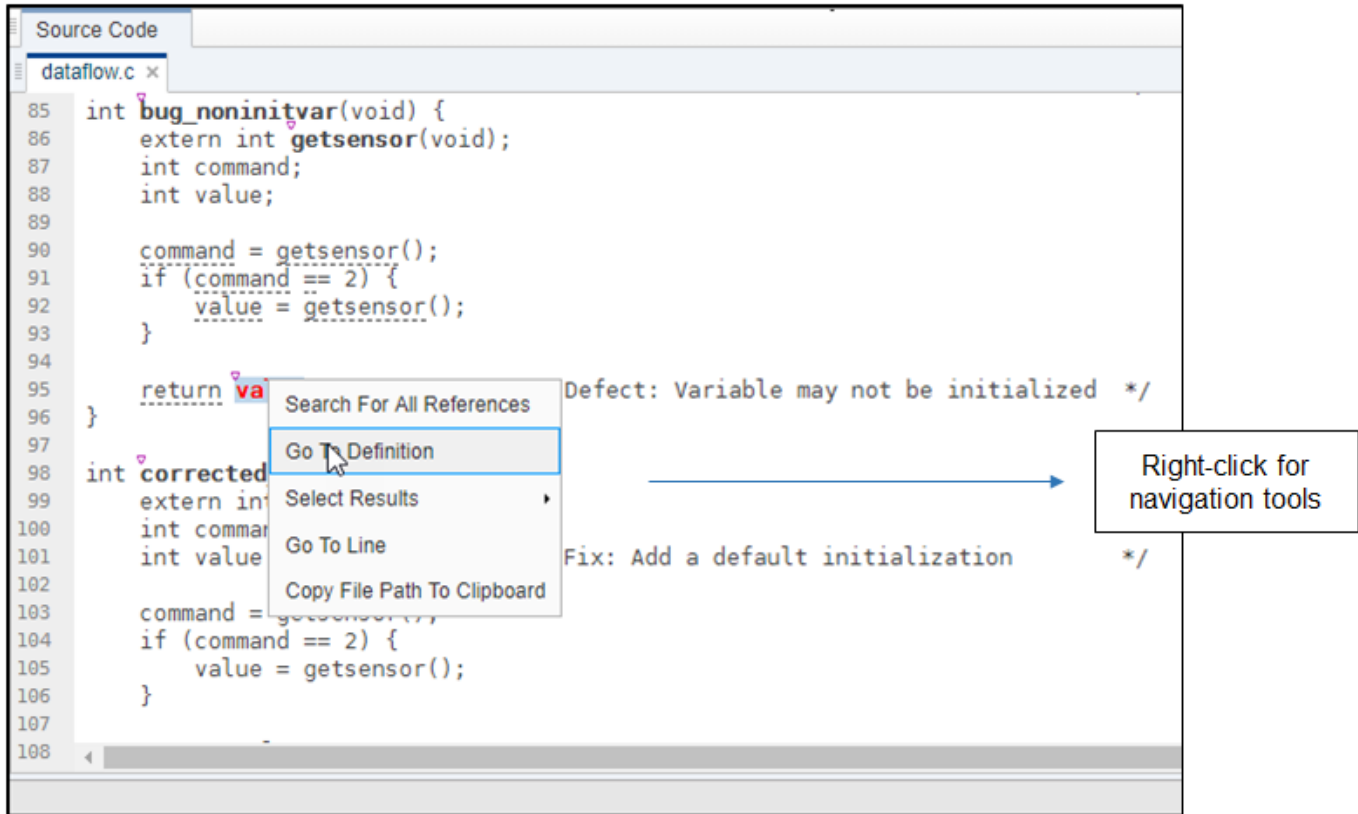
In the above event traceback, this sequence is shown:

- 1 A variable `value` is declared.
- 2 The execution path bypasses an `if` statement. This information might be relevant if the variable is initialized inside the `if` block.
- 3 Location of the current defect: **Non-initialized variable**

Typically, the traceback shows major points in the control flow: entering or bypassing conditional statements or loops, entering a function, and so on. For specific defects, the traceback shows other kinds of events relevant to the defect. For instance, for a **Declaration mismatch** defect, the traceback shows the two locations with conflicting declarations.

Create Your Own Navigation Path

If the event traceback is not available, use other navigation tools to trace your own path through the code.




Before you begin navigating through pathways in your code, ask the question: What am I looking for? Based on your answer, choose the appropriate navigation tool. For instance:

- To investigate a **Non-initialized variable** defect, you might want to make sure that the variable is not initialized at all. To look for previous instances of the variable, on the **Source Code** pane, right-click the variable and select **Search For All References**. This option lists only instances of a specific variable and not other variables with the same name in other scopes.
- To investigate a violation of **MISRA C:2012 Rule 17.7**:

The value returned by a function having non-void return type shall be used.

you might want to navigate from a function call to the function definition. Right-click the function and select **Go To Definition**.

After you navigate away from the current result, use the  icon on the **Result Details** pane to come back.

To select a different result from the **Source Code** pane, Ctrl-click the result or right-click and select **Select Results At This Location**. The **Results Details** pane updates but the result you select is not highlighted in the **Results List** pane. Clicking a result in the **Results List** updates the **Results Details** and **Source Code** panes.

See Also

More About

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 26-2
- “Filter and Sort Results in Polyspace Access Web Interface” on page 27-8

Investigate the Cause of Empty Results List

This topic shows how to interpret an empty results list in the Polyspace Access web interface. To see how to interpret a similar empty list in the user interface of the Polyspace desktop products, see “Investigate the Cause of Empty Results List” on page 21-7.

When you review results from a Polyspace Bug Finder or Polyspace Bug Finder Server analysis, the **Results List** pane can be empty or it can display this message:

No results available for currently selected filters,
or no results available for the selected project.

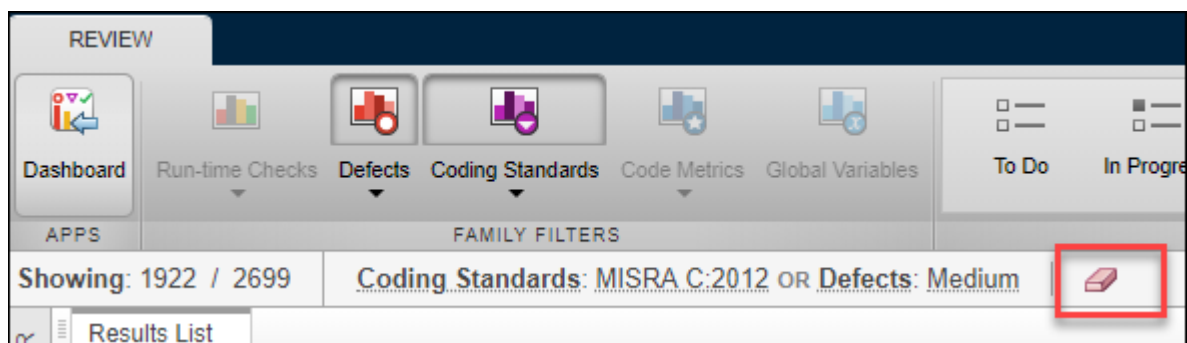
The message can indicate that your code has no defect or coding rule violation. However, before you reach this conclusion:

- 1 Open the **Run Log** pane by going to **Window > Run Log**.
- 2 Maximize the pane by double-clicking the **Run Log** tab, then use CTRL - F to check for the following.
 - Did all your source files compile?
Search for Failed compilation

If a file does not compile, Bug Finder can return some results, but only files with no compilation errors are fully analyzed.
 - Did you include all your source files in your project?
Search for verifying sources ...

Make sure that all the files that you want to analyze are listed under this message.
- 3 Open the **Configuration Used** pane by going to **Window > Configuration Used**, then:
 - Verify that the appropriate options are activated to check for coding standards violations and to compute code metrics.
 - Check if the `-fast-analysis` option is activated. Bug Finder checks for only a subset of defects and coding rules in fast analysis mode.
 - Click **Checkers** to see a list all the defects and coding rules checkers selected for this analysis.
- 4 Check whether you are applying any filters to the results.

To see which filters you are applying to the results, see the filter bar below the **FAMILY FILTERS** section of the toolbar. To clear all applied filters, click the eraser icon.



If you review results for an analysis you did not configure, discuss the possible causes of an empty results list with the project buildmaster. If you use `polyspace-configure` as part of your analysis workflow, the **Run Log** and **Configuration Used** panes might not contain all the analysis configuration parameters. For more information on analysis options and project configuration, see the documentation for Polyspace Bug Finder or Polyspace Bug Finder Server.

See Also

More About

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 26-2

Dashboard in Polyspace Access Web Interface

This topic focuses on the Polyspace Access web interface. To learn about the equivalent pane in the Polyspace desktop user interface, see “Dashboard in Polyspace Desktop User Interface” on page 21-9.

The **DASHBOARD** perspective provides an overview of the analysis results in graphical format, with clickable fields that let you drill down into your findings by project, file, or category.

When you upload an analysis run to the Polyspace Access database, the **DASHBOARD** updates to display the statistics for the latest run.

The screenshot displays the Polyspace Access Web Interface Dashboard. The interface is divided into several sections:

- Project Explorer:** A sidebar on the left showing a list of projects (project-84 to project-99) and folders (Examples-long-list-of-subprojects, Examples-pre18b, Examples-Trends). The selected project is "Bug_Finder_Example-Trends (Bug Finder)".
- Project Overview:** The main dashboard area showing a summary of statistics for the selected project.
 - Open Results:** 2,477 Open, 2,477 New, 0 Assigned To Me, 2,475 Unassigned.
 - Polyspace:** 26% progress, 2,278 Remaining, Exhaustive Threshold.
 - Code Metrics:** 0 Sub Projects, 3,315 Uncommented, 12 Files, 694 Cyclomatic.
 - Defects:** 77 Density, 254 Open, 254 New. Legend: To Do (254), In Progress (0), Done (158).
 - Coding Standards:** 671 Density, 2,223 Open, 2,223 New. Legend: To Do (2223), In Progress (0), Done (807).
- Trends:** A line graph titled "Open findings over time" showing the number of open findings from 12/23/2021 8:33:48 to 12/23/2021 9:44:40. The y-axis ranges from 0 to 2,500.
- Details:** A table showing the breakdown of findings by category.

Name	Total	To Do	In Progress	Done
Defects	412	254	–	158
Coding Standards	3,030	2,223	–	807

On the **Project Overview** dashboard, you see statistics for the currently selected project. When you select a folder in the **Project Explorer**, you see an aggregate of statistics for all the projects under that folder. The aggregate does not include the statistics of projects for which you do not have a role of **Administrator**, **Owner**, or **Contributor**.

In the **Summary** section of the **Project Overview** dashboard, cards display information about open issues, code metrics, quality objectives, and the different families of findings.

- The **Run-time Check** card (Code Prover) shows a distribution of findings as red, orange, gray, and green. The card also shows the **Selectivity**, the number of green checks as a percentage of all detected run-time checks.
- **Defects** and **Coding Rules** cards show a distribution of findings as:
 - **To Do** — Findings with a status of `Unreviewed` that need to be addressed with a fix or a justification.
 - **In Progress** — Findings with a status of `To fix`, `To investigate`, or `Other` that need to be addressed with a fix or a justification.
 - **Done** — Findings with a status of `Justified`, `No action planned`, or `Not a defect`.

The card also shows the **Density**, the number of **To Do** and **In Progress** defects or coding standard violations per one thousand lines of code without comments. To view the density you must enable **Code Metrics** in your analysis.

Note Green run-time checks, green shared variables, non-shared variables, and code metrics do not need to be addressed or justified. These findings do not count toward the number of findings that are **To Do**, **In Progress**, and **Done**.

To see a more in-depth overview for a family of findings, open additional dashboards by clicking the corresponding card title in the **Project Overview** dashboard or by using the **DASHBOARDS** section of the toolstrip.

In the additional dashboards:

- The **Summary** section displays project statistics for that family of findings such as current progress of results review or code coverage information.
- The **Details** section displays a table that allows you to drill down into the findings by category or by file. If you select a folder that contains multiple projects, you see a categorization by project instead of by file.

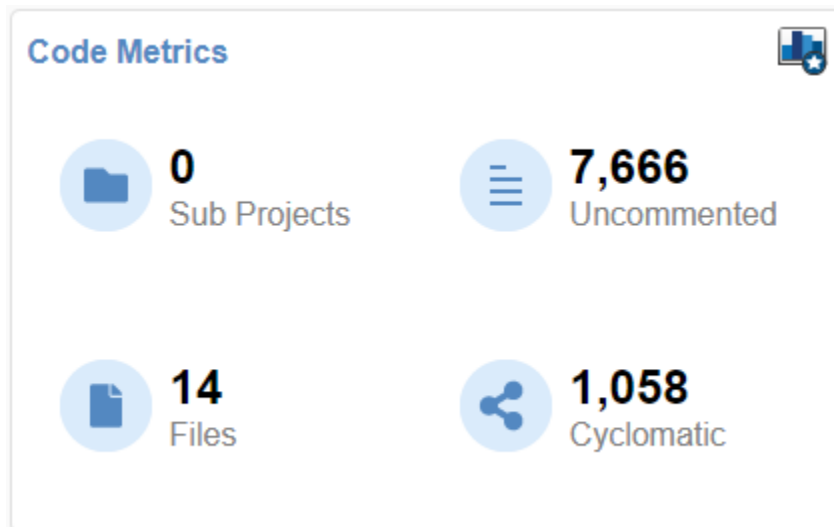
You can also perform these actions on the **DASHBOARD** perspective:

- View statistics for a previous run or compare a current run to a previous run. See “Compare Results in Polyspace Access Project to Previous Runs and View Trends” on page 27-19.
- Click elements on the graphs or tables to filter results from the **Results List** pane. See “Filter and Sort Results in Polyspace Access Web Interface” on page 27-8.
- Define and set quality objective levels. See “Quality Objectives Dashboard in Polyspace Access” on page 25-14.
- Manage projects and user authorizations. See “Manage Permissions and View Project Trends in Polyspace Access Web Interface” on page 27-2.
- Open the current project findings in the Polyspace desktop interface.

Code Metrics Dashboard in Polyspace Access Web Interface

To view the code complexity metrics that Polyspace computes, use the **Code Metrics** dashboard. See “Code Metrics”.

Polyspace computes the code complexity metrics during an analysis only when you use the option Calculate code metrics (-code-metrics).



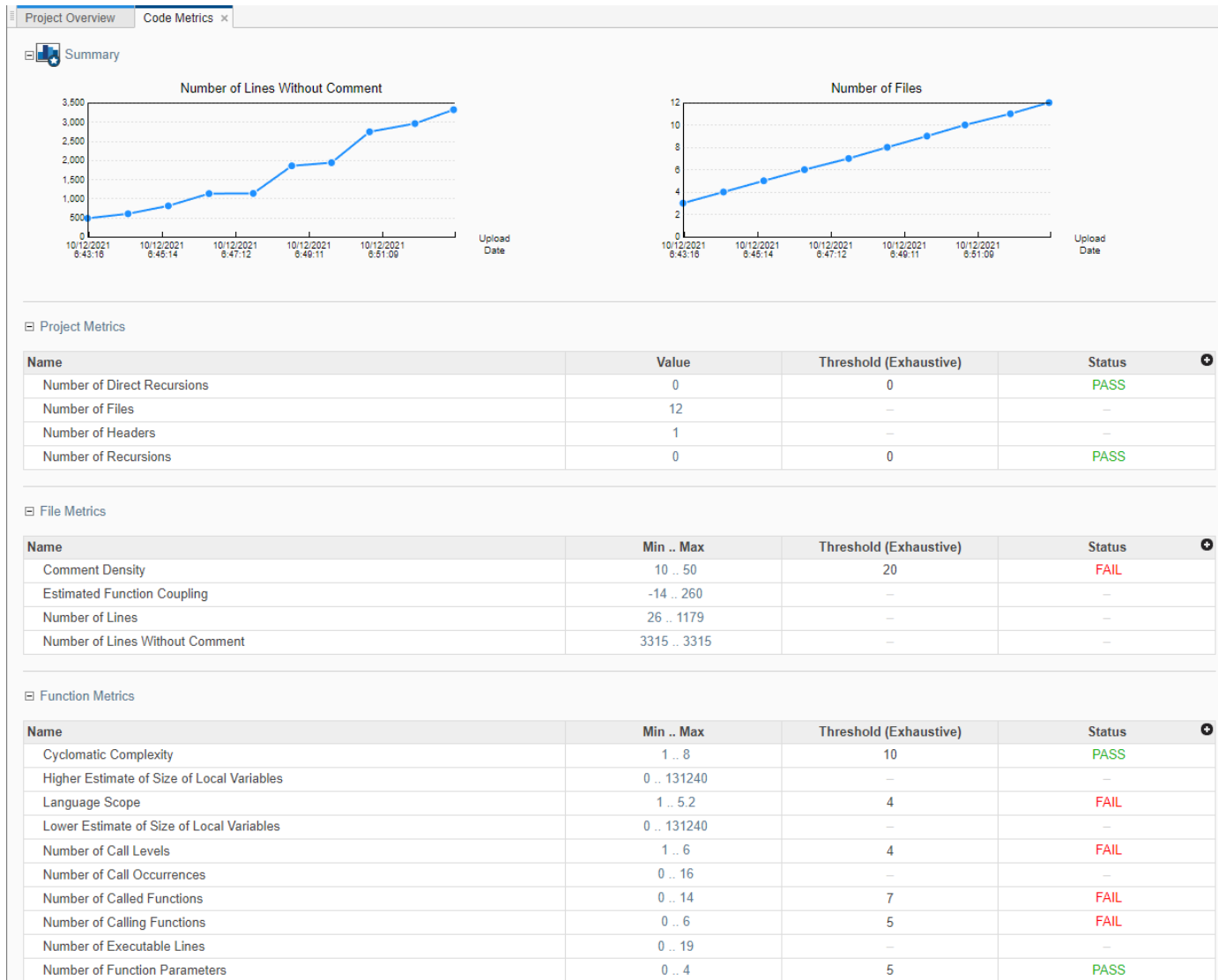
In the **Project Explorer**, select a project. Use the **Code Metrics** card in the **Project Overview** dashboard to get a quick overview of these code metrics:

- Number of Files (**Files**)
- Number of Lines Without Comment (**Uncommented**)
- Cyclomatic Complexity (**Cyclomatic**)

If you select a folder in the **Project Explorer**, the **Code Metrics** card shows:

- The number of **Sub Projects** in that folder. This number includes only subprojects that are directly (one level) below the top level folder.
- An aggregate of the other metrics on the card for all the subprojects at any level for which you are a **Contributor**, an **Owner**, or an **Administrator**.

To open the **Code Metrics** dashboard, click the **Code Metrics** icon in the **DASHBOARD** section of the toolstrip. Or, click **Code Metrics** on the card in the **Project Overview** dashboard.



In the **Summary** section, you see trend charts of the **Number of lines Without Comment** and **Number of Files** for the project.

The other sections of the dashboard display tables with the computed value or range of the different project, file, and function metrics. When applicable, the table shows the predefined threshold and pass/fail status for the corresponding code metric. For a list of code complexity metrics thresholds, see “HIS Code Complexity Metrics” on page 16-54. If you select a folder in the **Project Explorer**, the tables in the **Code Metrics** dashboard do not show the threshold or pass/fail status. The value or range of the metrics are aggregate of all subprojects in the selected folder. To drill down to a project from this aggregate view, expand a table row and click the project name.

To improve your code quality, use the pass/fail status to identify and lower metrics values that exceeds a threshold.

For instance, if the **Number of Called Functions** range exceeds the predefined threshold:

- 1** Click **FAIL** in the **Status** column or click the range in the **Min..Max** column to open the **Results List** filtered to the **Number of Called Functions** metric
- 2** Review the results that exceed the metric threshold. If several of those functions are always called together, you can write one function that fuses the bodies of those functions. Call that one function instead of the group of functions that are called together.

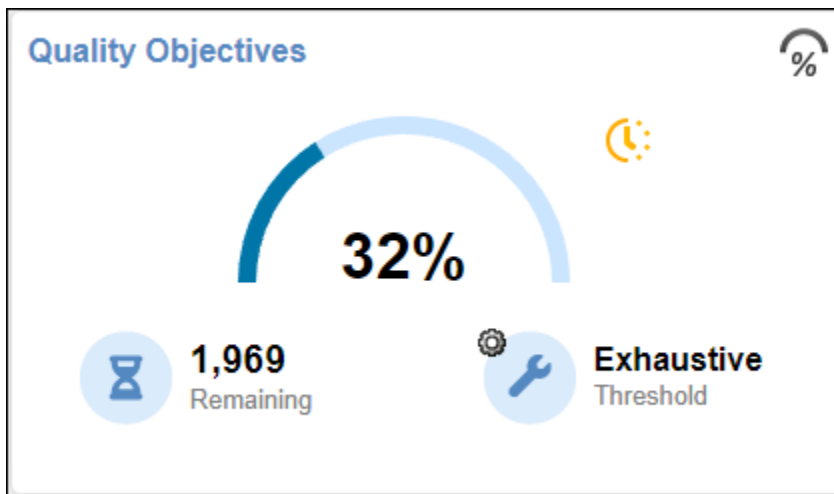
Quality Objectives Dashboard in Polyspace Access

To monitor the quality of your code against predefined on page 31-2 software quality thresholds or user-defined thresholds, use the **Quality Objectives** dashboard. You can use the thresholds as pass or fail criteria during the various stages of your project. From the dashboard, you can:





- Apply the default on page 31-2 Polyspace Access quality objectives or create custom objectives that suit your requirements. See “Customize Software Quality Objectives” on page 25-16.
- view a snapshot of your code quality against all levels of the currently applied quality objectives definition.

To manage the thresholds that you assign to projects, see “Manage Software Quality Objectives in Polyspace Access” on page 27-13.


Monitor Code Quality Against Software Quality Objectives




In the **Project Overview** dashboard, use the **Quality Objectives** card to get a quick overview of your progress in achieving a quality objective threshold. The card shows:


- The percentage of findings already addressed to achieve the selected threshold.
- One of these labels:
 -  (pass) — All findings for this threshold have been addressed.
 -  (in progress) — Some findings for this threshold are still open. A finding is open if it has a review status of Unreviewed, To fix, To investigate, or Other.
 -  (incomplete) — Some checkers required for this threshold were not activated in the analysis. For instance, if a threshold requires that you address all Polyspace Bug Finder defects, but the analysis includes only **Numerical** defects, the level is incomplete, even if you address all findings. To see a list of checkers you must activate, click .


Note This label applies only to SQO thresholds 1 through 6. If you select the **Exhaustive**

software quality threshold and you address all the findings, the threshold is labeled as (pass) even if all required checkers for this threshold were not activated in the analysis. 

 (not computed) — No quality objective results were computed.

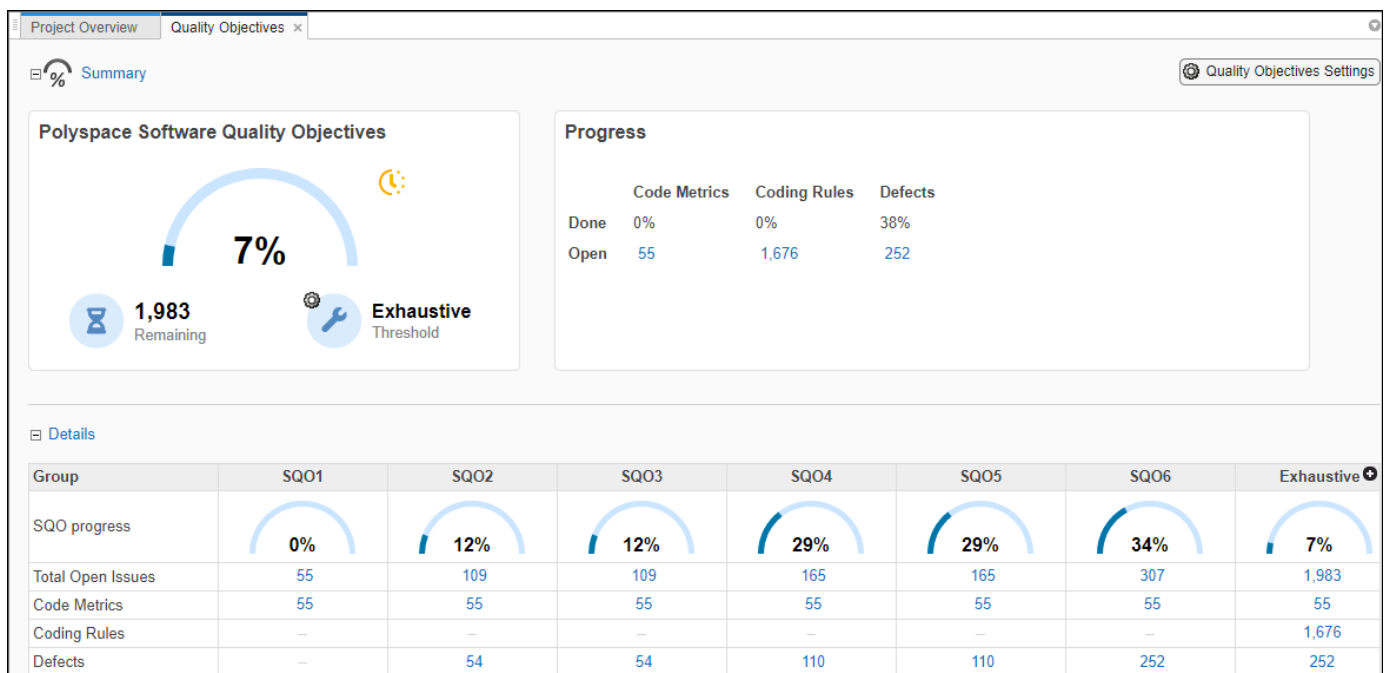
After you apply a new quality objectives definition to a project, you see the not computed label until you upload a new run to the project.

- The assigned **Threshold**. To select a different threshold or quality objectives definition, click . You must be an **Administrator** or project **Owner** to assign quality objective definitions or thresholds to a project. You can also assign quality objectives by right-clicking a project in the **Project Explorer**.
- The **Remaining** number of findings that you need to address to reach the threshold. Click this number to open the **Review** perspective and see these findings in the **Results List**.

For a more comprehensive view, open the **Quality Objectives** dashboard. In the **Summary** section, click  in the card on the left to pick a threshold and see the remaining open issues, including a breakdown for each category, such as code metrics or coding rules.

In this **Quality Objectives** dashboard, 7% of the findings required to achieve threshold **Exhaustive** have been addressed, include 38% of **Defects**. There are 1983 open issues, which are split between **Code Metrics** (55), **Coding Rule** (1676), and **Defects** (252).

This table shows the current progress of code quality for all quality objective thresholds. To view the **Results List** for a set of open issues, click the corresponding value in the table.



Additionally, you can view aggregated SQO statistics by selecting an entire project folder in the **Project Explorer** pane. Note these differences between viewing SQO information at the folder level and the project level.

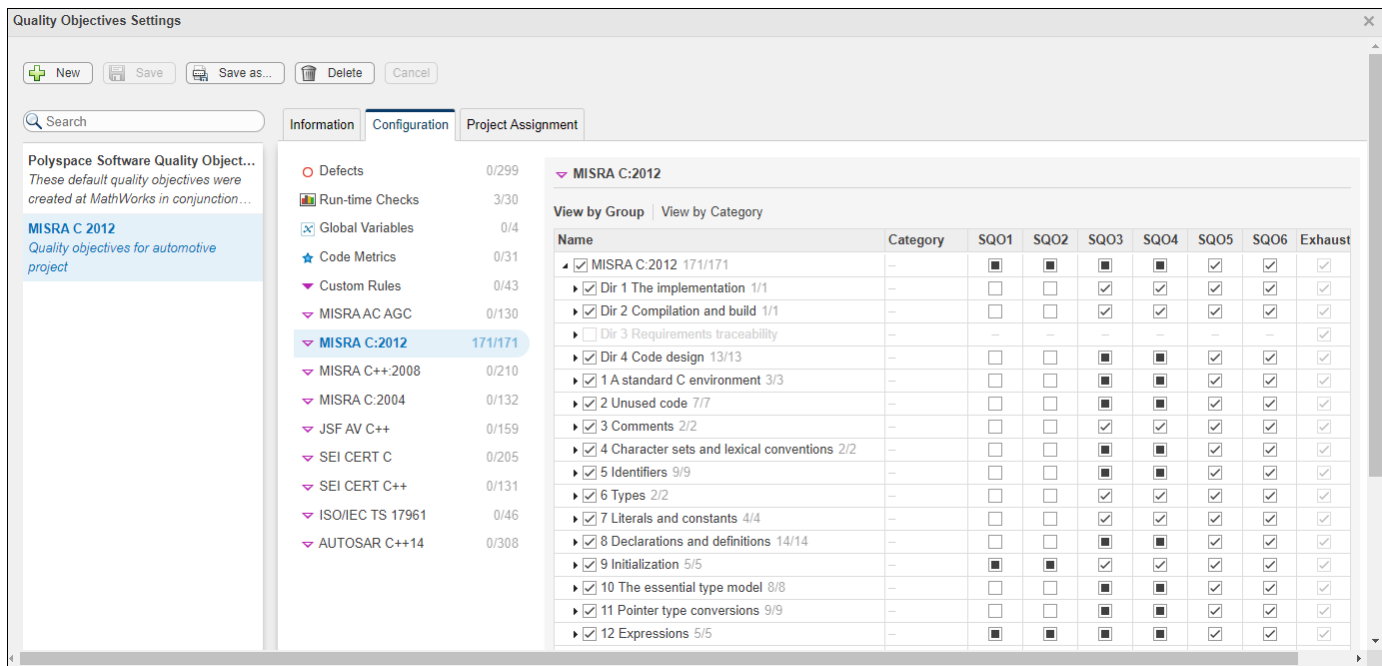
- For folders, the **Progress** and **Details** sections do not contain links to filtered results in the tables.
- You cannot assign quality objective levels to all projects in the folder at once.
- The **Quality Objectives** card for folders does not show **Pass**, **In progress**, or **Incomplete** labels.

Polyspace Access aggregates SQO statistics even if the quality objectives configuration levels are not homogeneous. While individual projects might contain separate definitions of each SQO level, Polyspace Access does not separate the statistics by level details when aggregating the statistics. All SQO1 level projects are aggregated together, as are SQO2 and so forth.

Customize Software Quality Objectives

To customize the thresholds that you use as pass or fail criteria to track the quality of your code, create or edit quality objective definitions and apply these definitions to specific projects. For instance, you might have a project where you want to check the quality of your code against only the MISRA C:2012 coding standard.

To open the quality objectives settings, click **Quality Objectives Settings** on the **Quality Objectives** dashboard.



Create Quality Objectives Definition

To create a quality objectives definition, click **New** and enter a name for the new definition. You can optionally provide a description for the quality objectives definition and for the different SQO levels of that definition. Go to the **Information** tab to view or make additional edits to the descriptions.

After you assign this definition to a project, the name of the definition is displayed on the card in the summary section of the **Quality Objectives** dashboard for that project.

Edit Quality Objectives Definition

You can edit quality objective definitions only if you have a Polyspace Access role of **Administrator** or **Owner**. To set user roles, see “Manage Project Permissions” on page 27-3.

This table lists the different Polyspace Access roles and their corresponding write permissions for the quality object definitions.

Project Role	Write Permission
Administrator	You can edit any quality objective definition.
Owner	You can edit the quality objective definitions that you created.
Contributor	you have a read-only view of the quality objective settings and cannot make edits.

You cannot edit the default **Polyspace Software Quality Objectives**, no matter your role.

To edit the thresholds selection for a quality objectives definition:

- 1 Select the definition in the left pane of the **Configuration** tab.
- 2 Click a findings family, for instance, MISRA C:2004.

To choose individual results, select or expand the nodes. For each family of results, you can view the nodes by group, or by category when available.

When you select nodes in the leftmost part of the table:

- indicates that all entries under the node are enabled.
- indicates that some entries under the node are not enabled.

For the quality objective thresholds under the SQO columns:

- indicates that all the entries that are enabled under the node on that row apply to this threshold.
- indicates that some of the entries that are enabled under the node on that row do not apply to this threshold.

	Category	SQO1	SQO2	SQO3	SQO4	SQO5	SQO6	Exhaus
▲ <input checked="" type="checkbox"/> MISRA C:2004 52/131	–	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
▶ <input type="checkbox"/> 1 Environment 0/1	–	–	–	–	–	–	–	<input checked="" type="checkbox"/>
▲ <input checked="" type="checkbox"/> 2 Language extensions 2/3	–	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<input type="checkbox"/> 2.1 Assembly language shall be encaps...	Required	–	–	–	–	–	–	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> 2.2 source code shall only use /* ... */ st...	Required	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> 2.3 The character sequence /* shall not ...	Required	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

For example, in the previous figure, looking at the **Language extensions** node:

- The leftmost part of the table is marked as because rule 2.1 is not enabled.
- The SQO5 and SQO6 entries along the row of the node are marked as because all the rules that are enabled under the node apply to these SQO thresholds.

- The SQO4 entry along the row of the node is marked as because rule 2.2 is enabled but does not apply to this SQO threshold.

These results are customizable by specifying numerical inputs:

- **Run-time Checks** — Customize the percentage of findings that you must address or justify for each threshold. Enter a value between 0 and 100. To disable the selection, leave the entry blank.
- **Code Metrics** — Customize the value of the different metrics for each threshold. To disable the selection, leave the entry blank.

When you make a selection for a threshold, all higher thresholds inherit that selection. For instance, if you select a coding rule for SQO3, the rule is also selected for SQO4, SQO5, and SQO6. By default, when you first select a node or an individual result, the selection applies only to SQO6.

To save your changes, click **Save**. You can also edit a quality objective definition by creating a copy of the definition using the **Save as** button and making edits to that copy. You might want to create a copy if:

- You do not have write permissions for a quality objective definition.
- You want to edit a quality objective definition but apply the changes to only your project.
- You want to use an existing definition as a template.

If you make changes to a quality objectives definition that applies to multiple projects, Polyspace Access displays a warning with a link to the **Project Assignment** tab on the **Quality Objectives Settings** window. Open the tab to determine which projects are affected by your changes and inform users that have access to those projects of your changes.

See Also

More About

- “Evaluate Polyspace Bug Finder Results Against Bug Finder Quality Objectives” on page 31-2
- “Code Metrics”

Results List in Polyspace Access Web Interface


This topic focuses on the Polyspace Access web interface. To learn about the equivalent pane in the Polyspace desktop user interface, see “Results List in Polyspace Desktop User Interface” on page 21-15.

The **Results List** pane lists all results along with their attributes.

For each result, the **Results List** pane contains the result attributes, listed in columns:


Attribute	Description
Family	Group to which the result belongs.
ID	Unique identification number of the result.
Type	Defect or coding rule violation.
Group	Category of the result, for instance: <ul style="list-style-type: none"> For defects: Groups such as static memory, numerical, control flow, concurrency, etc. For coding rule violations: Groups defined by the coding rule standard. For instance, MISRA C: 2012 defines groups related to code constructs such as functions, pointers and arrays, etc.
Check	Result name, for instance: <ul style="list-style-type: none"> For defects: Defect name For coding rule violations: Coding rule number
Information	Result sub-type when available. <ul style="list-style-type: none"> For defects: Impact classification. For coding standards: required or mandatory, rule or recommendation.
Detail	Additional information about a result. The column shows the first line of the Result Details pane. For an example of how to use this column, see the result MISRA C:2012 Dir 1.1.
File	File containing the instruction where the result occurs
Function	Function containing the instruction where the result occurs. If the function is a method of a class, it appears in the format <code>class_name::function_name</code> .

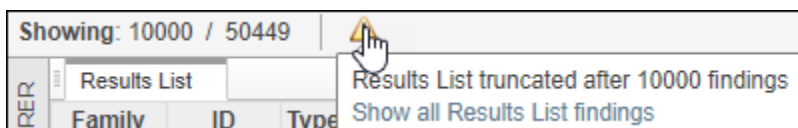
Attribute	Description
Status	Review status you have assigned to the result. The possible statuses are: <ul style="list-style-type: none"> • Unreviewed (default status) • To investigate • To fix • Justified • No action planned • Not a defect • Other See also “Add Custom Status in Polyspace Access Project” on page 26-3.
Severity	Level of severity you have assigned to the result. The possible levels are: <ul style="list-style-type: none"> • Unset • High • Medium • Low
Assigned to	User name of reviewer assigned to this result.
Ticket Key	When you create a bug tracking tool (BTT) ticket for a result, this field contains the ticket ID. Click the ticket ID in the Results Details to open the ticket in the BTT interface.
Comments	Comments you have entered about the result
Folder	Path to the folder that contains the source file with the result

To show or hide any of the columns, click the  icon in the upper-right of the **Results List** pane, then select or clear the title of the column that you want to show or hide.

Using this pane, you can:

- Navigate through the results.
- Organize your result review using filters in the toolstrip or in the context menu. For more information, see “Filter and Sort Results in Polyspace Access Web Interface” on page 27-8.
- Right-click a result to get the URL of the result. When you open this URL in a web browser you get see the **Results List** pane filtered to that one result.

If the **Results List** exceeds 10000 findings, Polyspace Access truncates the list and displays this icon  in the filters bar. To show all findings, see the contextual help of the icon.

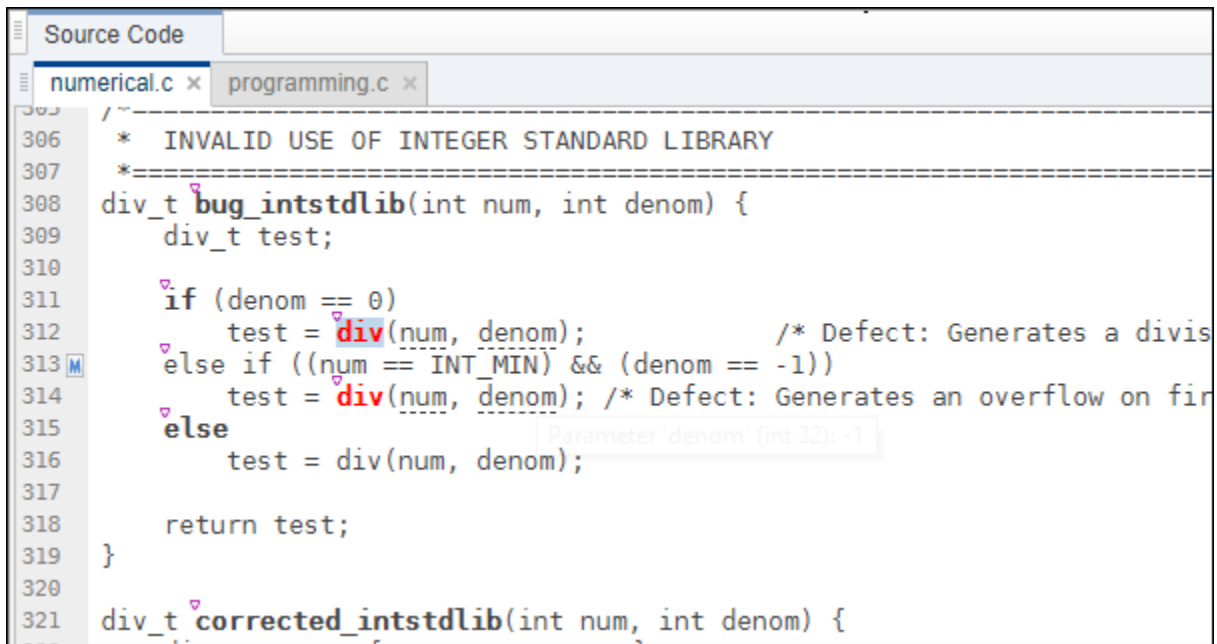


The 10000 findings limit is preset and cannot be changed.

Source Code in Polyspace Access Web Interface

This topic focuses on the Polyspace Access web interface. To learn about the equivalent pane in the Polyspace desktop user interface, see “Source Code in Polyspace Desktop User Interface” on page 21-17.

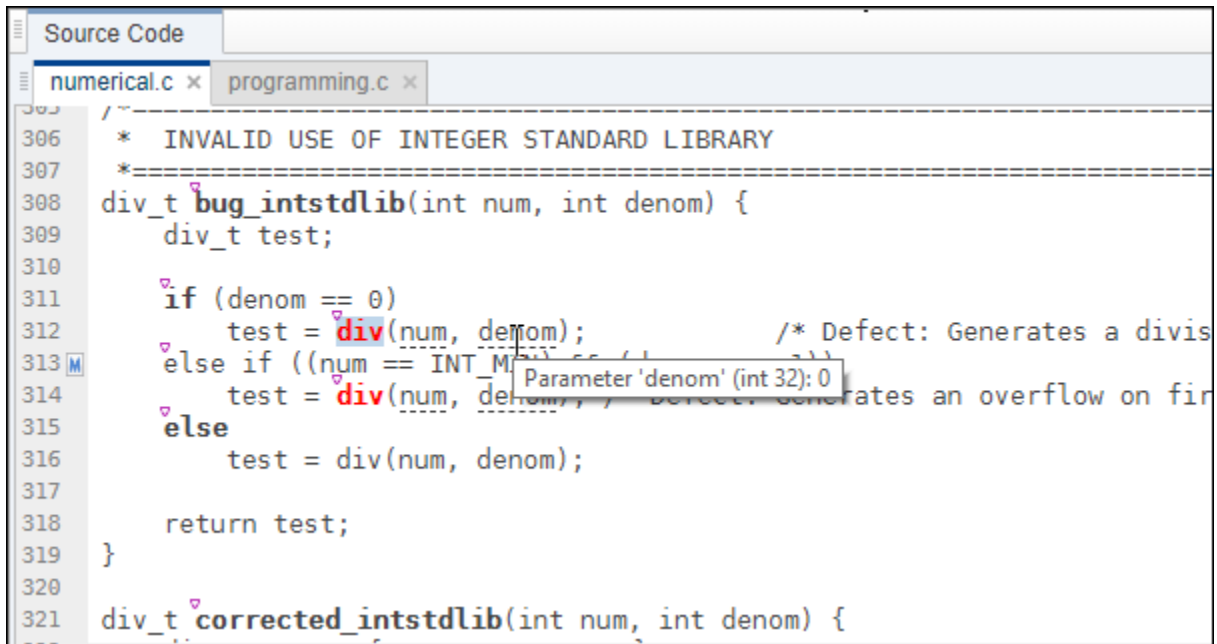
The **Source Code** pane shows the source code with the defects colored in red.



```
Source Code
numerical.c x programming.c x
306  /*
307  *  INVALID USE OF INTEGER STANDARD LIBRARY
308  *  -----
309  div_t bug_intstdlib(int num, int denom) {
310      div_t test;
311
312      if (denom == 0)
313          test = div(num, denom); /* Defect: Generates a divis
314      else if ((num == INT_MIN) && (denom == -1))
315          test = div(num, denom); /* Defect: Generates an overflow on fir
316      else
317          test = div(num, denom);
318
319      return test;
320  }
321  div_t corrected_intstdlib(int num, int denom) {
```

Tooltips

Placing your cursor over a result displays a tooltip that provides range information for variables, operands, function parameters, and return values.



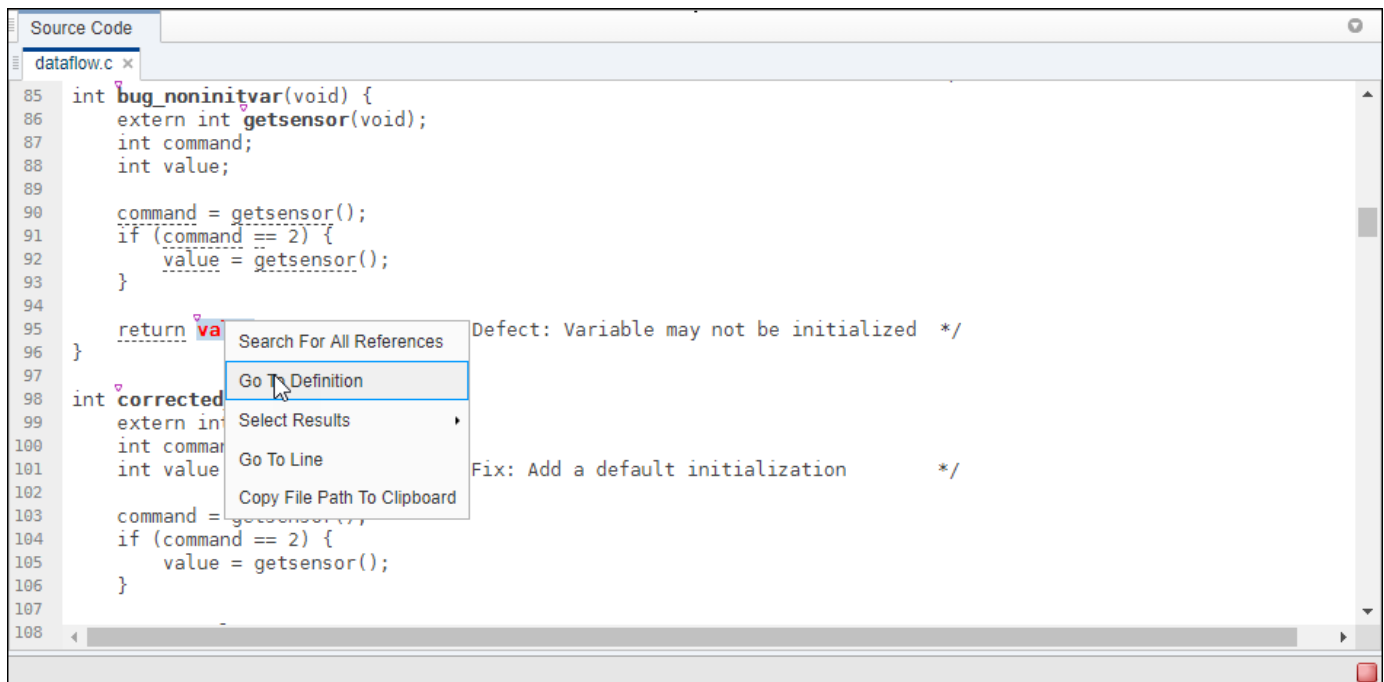
```

Source Code
numerical.c x programming.c x
305 /-----
306  * INVALID USE OF INTEGER STANDARD LIBRARY
307  *-----
308  div_t bug_intstdlib(int num, int denom) {
309      div_t test;
310
311      if (denom == 0)
312          test = div(num, denom); /* Defect: Generates a divis
313  else if ((num == INT_M
314          test = div(num, denom); /* Defect: Generates an overflow on fir
315  else
316          test = div(num, denom);
317
318      return test;
319  }
320
321  div_t corrected_intstdlib(int num, int denom) {

```

Examine Source Code

On the **Source Code** pane, if you right-click a text string, the context menu provides options to examine your code:




```

Source Code
dataflow.c x
85  int bug_noninitvar(void) {
86      extern int getsensor(void);
87      int command;
88      int value;
89
90      command = getsensor();
91      if (command == 2) {
92          value = getsensor();
93      }
94
95      return value; /* Defect: Variable may not be initialized */
96  }
97
98  int corrected
99      extern in
100      int comman
101      int value
102
103      command =
104      if (command == 2) {
105          value = getsensor();
106      }
107
108

```

For example, if you right-click the variable, you can use the following options to examine and navigate through your code:


- **Search For All References** — List all references in the **Code Search** pane. The software supports this feature for global and local variables, functions, types, and classes.
- **Go To Definition** — Go to the line of code that contains the definition of `i`. The software supports this feature for global and local variables, functions, types, and classes. If a definition is not available to Polyspace, selecting the option takes you to the declaration.
- **Select Results** -- Show more information about the selected result in the **Results Details** pane and pin the result in the **Source Code** pane.

After you navigate away from the current result, use the  icon on the **Result Details** pane to come back.

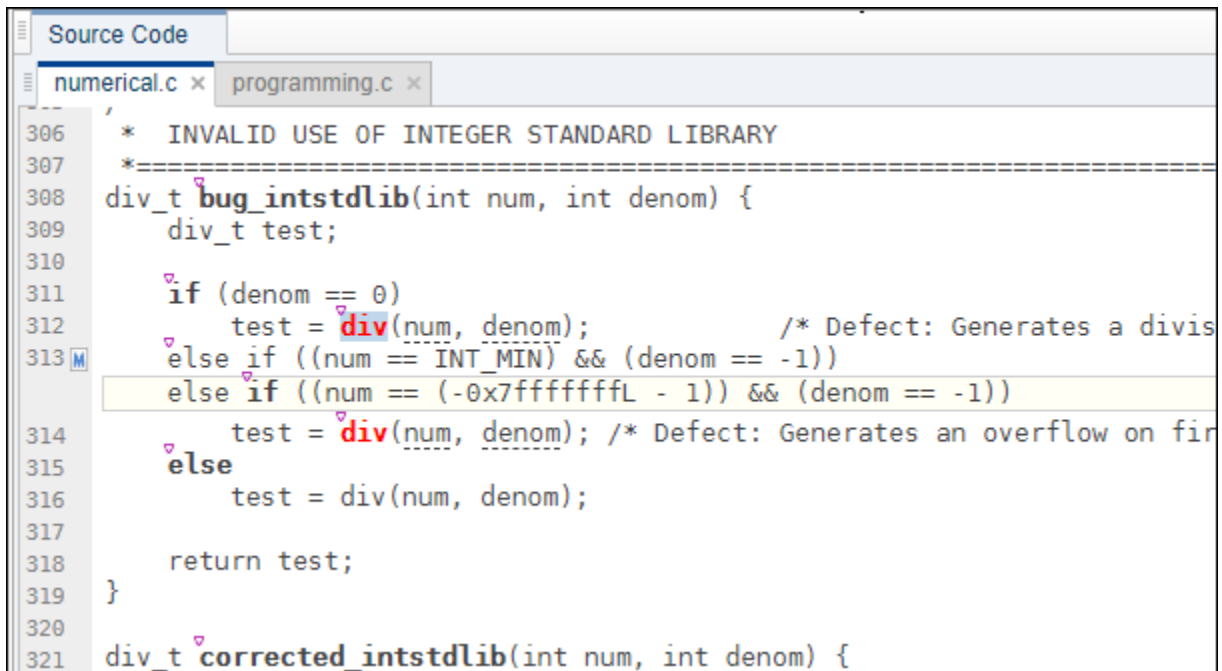
- **Go To Line** — Open the Go to line dialog box. If you specify a line number and click **Enter**, the software displays the specified line of code.

To search for instances of your selection in the **Current Source File** or in **All Source Files**, double-click your selection before you right-click.

Expand Macros

You can view the contents of source code macros in the source code view. A code information bar displays  icons that identify source code lines with macros.

When you click this icon, the software displays the contents of macros on the next line.



```

Source Code
numerical.c x programming.c x
306 * INVALID USE OF INTEGER STANDARD LIBRARY
307 *=====
308 div_t bug_intstdlib(int num, int denom) {
309     div_t test;
310
311     if (denom == 0)
312         test = div(num, denom); /* Defect: Generates a divis
313 M else if ((num == INT_MIN) && (denom == -1))
314         else if ((num == (-0x7fffffffL - 1)) && (denom == -1))
315             test = div(num, denom); /* Defect: Generates an overflow on fir
316         else
317             test = div(num, denom);
318     return test;
319 }
320
321 div_t corrected_intstdlib(int num, int denom) {

```

To display the normal source code again, click the icon again.

Note

- 1 The **Result Details** pane also allows you to view the contents of a macro if the check you select lies within a macro.

- You cannot expand OSEK API macros in the **Source Code** pane.

View Code Block

On the **Source Code** pane, to highlight a block of code, click either its opening or closing brace. If the brace itself is highlighted, click the brace twice.

```

306  * INVALID USE OF INTEGER STANDARD LIBRARY
307  *=====
308  div_t bug_intstdlib(int num, int denom) {
309      div_t test;
310
311      if (denom == 0)
312          test = div(num, denom);          /* Defect: Generates a divis
313  else if ((num == INT_MIN) && (denom == -1))
314          test = div(num, denom); /* Defect: Generates an overflow on fir
315  else
316          test = div(num, denom);
317
318      return test;
319  }
320
321  div_t corrected_intstdlib(int num, int denom) {

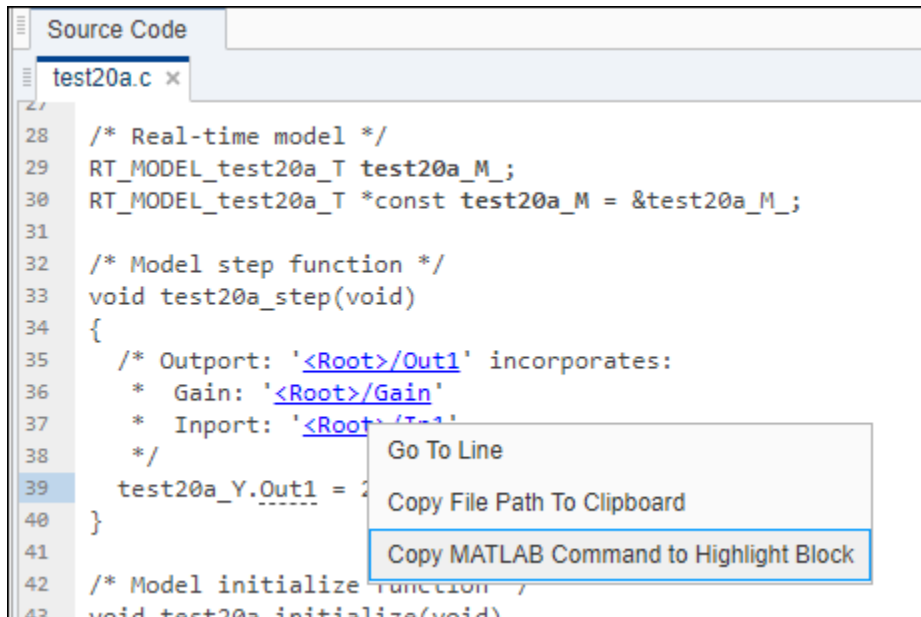
```

Navigate from Code to Model

If you run Polyspace on generated code in Simulink and upload the results to Polyspace Access, you can navigate from the source code in Polyspace Access to blocks in the model.

On the **Source Code** pane in the Polyspace Access web interface, links in code comments show blocks that generate the subsequent lines of code. To see the block in the model:

- Right-click a link and select **Copy MATLAB Command to Highlight Block**.



The screenshot shows a source code editor window titled "Source Code" with a sub-tab "test20a.c". The code is as follows:

```
28 /* Real-time model */
29 RT_MODEL_test20a_T test20a_M;
30 RT_MODEL_test20a_T *const test20a_M = &test20a_M;
31
32 /* Model step function */
33 void test20a_step(void)
34 {
35     /* Outport: '<Root>/Out1' incorporates:
36      * Gain: '<Root>/Gain'
37      * Inport: '<Root>/In1'
38      */
39     test20a_Y.Out1 = 2;
40 }
41
42 /* Model initialize function */
43 void test20a_initialize(void)
```

A context menu is open over line 39, with the following options:

- Go To Line
- Copy File Path To Clipboard
- Copy MATLAB Command to Highlight Block

This action copies the MATLAB command required to highlight the block. The command uses the Simulink.ID.hilite function.



- In MATLAB editor, paste and run the copied command with the model open.

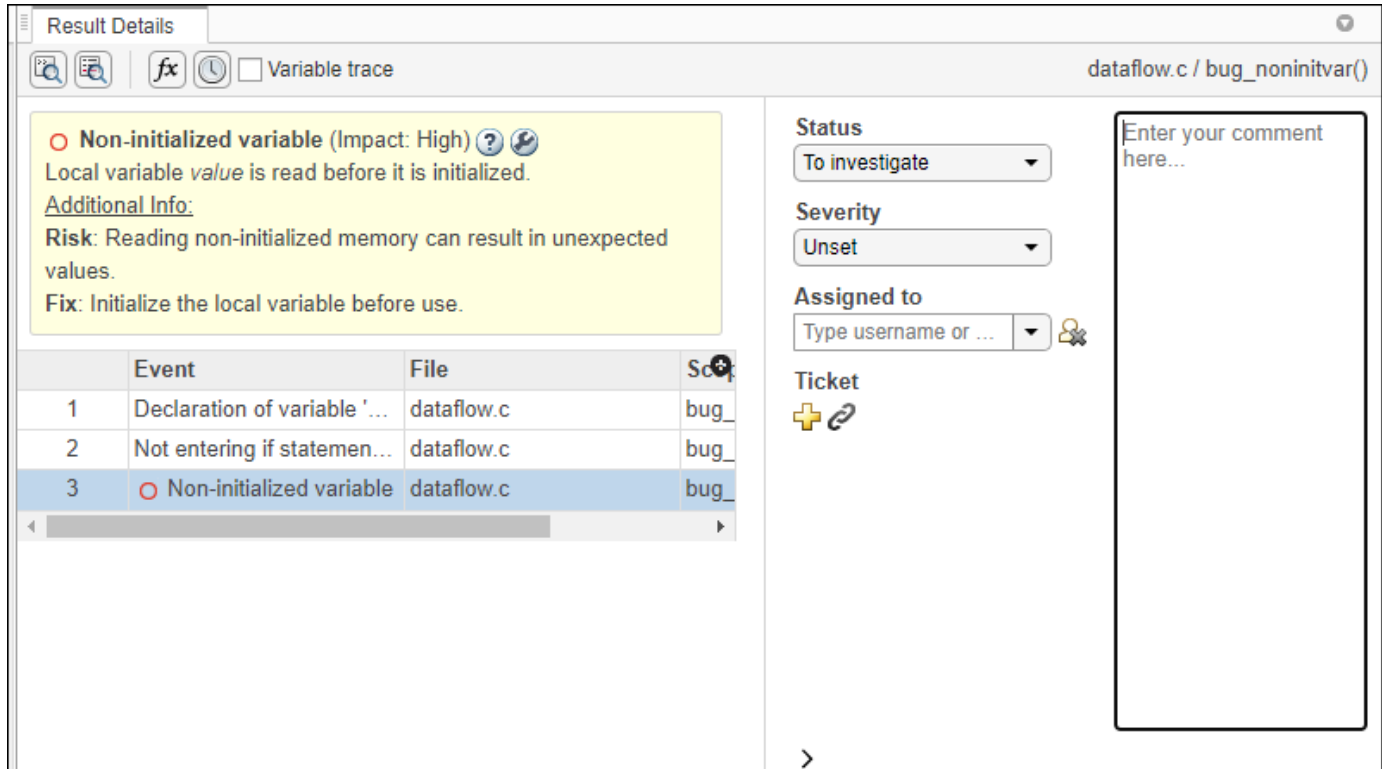
Result Details in Polyspace Access Web Interface

This topic focuses on the Polyspace Access web interface. To learn about the equivalent pane in the Polyspace desktop user interface, see “Result Details in Polyspace Desktop User Interface” on page 21-22.

The **Result Details** pane contains comprehensive information about a specific defect. To see this information, on the **Results List** pane, select the defect.

- The top right corner shows the file and function containing the defect, in the format *file_name/function_name*.
- The yellow box contains the name of the defect with an explanation of why the defect occurs.

The  button allows you to access documentation for the defect. When available, click the  icon to see fix suggestions for the defect.



The screenshot shows the 'Result Details' pane for a defect titled 'Non-initialized variable (Impact: High)'. The defect description states: 'Local variable *value* is read before it is initialized.' It includes a risk statement: 'Risk: Reading non-initialized memory can result in unexpected values.' and a fix suggestion: 'Fix: Initialize the local variable before use.' Below the description is a table of events:

	Event	File	Score
1	Declaration of variable '...	dataflow.c	bug_
2	Not entering if statemen...	dataflow.c	bug_
3	Non-initialized variable	dataflow.c	bug_




On the right side of the pane, there are configuration options: 'Status' (set to 'To investigate'), 'Severity' (set to 'Unset'), and 'Assigned to' (a dropdown menu). There is also a 'Ticket' section with a plus sign and a link icon. A large text area on the right is labeled 'Enter your comment here...'.

On this pane, you can also:

- Assign a **Severity** and **Status** to each check, and enter comments to describe the results of your review.
- Assign a reviewer to the result. A reviewer can filter the **Results List** to only show results that are assigned to him or her.
- Create a ticket in a bug tracking tool such as JIRA. Once you create the ticket the **Results Details** for this defect shows a clickable link to the ticket you created.
- View the event traceback.

The **Event** column lists the sequence of code instructions causing the defect. The **Scope** column lists the function containing the instructions. If the instructions are not in a function, the column lists the file containing the instructions.

The **Variable trace** check box allows you to see an additional set of instructions that are related to the defect.

- Click the  icon to open the “Call Hierarchy in Polyspace Access Web Interface” on page 25-28.
- Click the  icon to open the “Review History in Polyspace Access Web Interface” on page 25-33.
- Click the  icon to open the:

- **Error Call Graph** if the selected finding is a **Run-time Check**.

The pane displays the call sequence that leads to the detected finding. Click a node on the graph to navigate back to the source code.

- **Variable Access Graph** if the selected finding is a **Global variable**.

The pane displays a graphical representation of the access operations on global variables. Click a node on the graph to navigate back to the source code at the location of calling and called functions.

See Also



More About

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 26-2
- “Review History in Polyspace Access Web Interface” on page 25-33

Call Hierarchy in Polyspace Access Web Interface


This topic focuses on the Polyspace Access web interface. To learn about the equivalent pane in the Polyspace desktop user interface, see “Call Hierarchy in Polyspace Desktop User Interface” on page 21-24.

The **Call Hierarchy** pane displays the call tree of functions in the source code.

For each function `foo`, the **Call Hierarchy** pane lists the functions and tasks that call `foo` (callers) and those called by `foo` (callees). The callers are indicated by . The callees are indicated by . The **Call Hierarchy** pane lists direct function calls and indirect calls through function pointers.

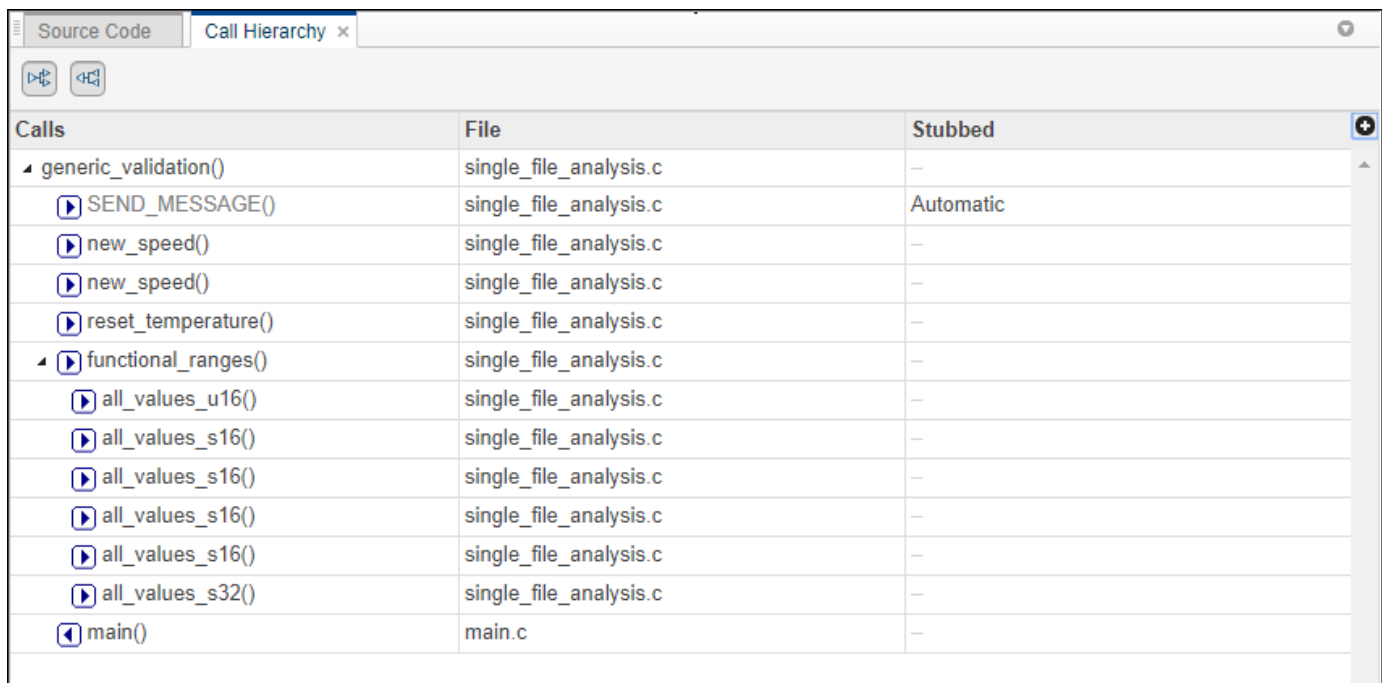
Note For Polyspace Access findings, you might not see all callers or callees of a function, especially for calls through function pointers and dead code.

For instance, Polyspace Access does not display the functions registered with `at_exit()` and `at_quick_exit()`, and called by `exit()` and `quick_exit()` respectively.

You open the **Call Hierarchy** pane by using the  icon in your **Results Details** pane, or by going to **Window > Call Hierarchy**.

To update the pane, click a defect on the **Results List** or CTRL-click a result in the **Source Code** pane. You see the function containing the defect with its callers and callees.

In this example, the **Call Hierarchy** pane displays the function `generic_validation`, and with its callers and callees.



Calls	File	Stubbed
▲ generic_validation()	single_file_analysis.c	–
▶ SEND_MESSAGE()	single_file_analysis.c	Automatic
▶ new_speed()	single_file_analysis.c	–
▶ new_speed()	single_file_analysis.c	–
▶ reset_temperature()	single_file_analysis.c	–
▲ ▶ functional_ranges()	single_file_analysis.c	–
▶ all_values_u16()	single_file_analysis.c	–
▶ all_values_s16()	single_file_analysis.c	–
▶ all_values_s16()	single_file_analysis.c	–
▶ all_values_s16()	single_file_analysis.c	–
▶ all_values_s16()	single_file_analysis.c	–
▶ all_values_s32()	single_file_analysis.c	–
◀ main()	main.c	–

Tip To navigate to the call location in the source code, select a caller or callee name

In the **Call Hierarchy** pane, you can perform these actions:

- **Show/Hide Callers and Callees**

Customize the view to display callers only or callees only. Show or hide callers and callees by clicking this button



- **Navigate Call Hierarchy**

You can navigate the call hierarchy in your source code. For a function, double-click a caller or callee name to navigate to the caller or callee definition in the source code.

- **Determine If Function Is Stubbed**

From the **Stubbed** column, you can determine if a function is stubbed. The entries in the column show why a function was stubbed.

- **Automatic:** Polyspace cannot find the function definition. For instance, you did not provide the file containing the definition.
- **Std library:** The function is a standard library function. You do not provide the function definition explicitly in your Polyspace project.
- **Mapped to std library:** You map the function to a standard library function by using the option `-code-behavior-specifications`.

Configuration Settings in Polyspace Access Web Interface

The **Configuration Used** pane displays the options and checkers that were enabled to generate the results in the currently selected project. The **Options** tab shows user-specified options and options that are enabled by default.

You open the **Configuration Used** pane by going to **Window > Configuration Used**.

Results List Configuration Used	
Options	Value
-author	MathWorks
-checkers	all
-code-metrics	true
-compiler	gnu4.6
-critical-section-begin	BEGIN_CRITICAL_SECTION:Cs10, acquire_sensor:Cs11, acquire_printer:Cs12, acquire_sensor2:Cs13, acquire_printer2:Cs14
-critical-section-end	END_CRITICAL_SECTION:Cs10, release_sensor:Cs11, release_printer:Cs12, release_sensor2:Cs13, release_printer2:Cs14
-custom-rules	
-date	05/05/2021
-entry-points	bug_datarace_task1, bug_datarace_task2, bug_datarace_task3, bug_datarace_task4, bug_deadlock_task1, bug_deadlock_task2, bug_doublelock_task, bug_doubleunlock_task, bug_badlock_task, bug_badunlock_task, bug_dataracestdlib_task1, bug_dataracestdlib_task2, bug_destroylocked_task, corrected_datarace_task1, corrected_datarace_task2, corrected_datarace_task3, corrected_datarace_task4, corrected_deadlock_task1, corrected_deadlock_task2, corrected_doublelock_task, corrected_doubleunlock_task, corrected_badlock_task, corrected_badunlock_task, corrected_dataracestdlib_task1, corrected_dataracestdlib_task2, corrected_destroylocked_task
-generate-integration-context-info	true
-lang	C
-misra3	all
-prog	Bug_Finder_Example-Trends
-results-dir	
-target	x86_64
-verif-version	BuildNumber-9

Click **Checkers** to see which checkers are enabled for:

- “Defects”.
- “Coding Standards”, for instance MISRA C: 2012.
- “Custom Coding Rules”.


Checkers is not available for a Code Prover project if no coding standard or custom coding rules are enabled.

Review History in Polyspace Access Web Interface

The **Review History** pane displays changes to the **Status**, **Severity**, or **Comment** for a finding. For each change to these review fields, you see a separate row with:

- The date and time of the change.
- The user name of the user who made the change.
- The review field that changed, for instance **Severity**.
- The original value of the review field.
- The new value of the review field.

All the changes that you make to the review fields of findings in the Polyspace desktop interface are shown in a single row after you upload these findings to Polyspace Access. The **Review History** pane does not display the user name of the user who made these changes.

You open the **Review History** pane by using the  icon in your **Results Details** pane, or by going to **Window > Review History**.

ENVIRONMENT		REVIEW		
Result Details		Review History x		
Show All				
Date and Time	User	What Chan	Original value	New value
4/27/2020 3:35:15 PM	ps_user	Comment	Reassigning to project owner	Changing severity to low
4/27/2020 3:35:04 PM	ps_user	Severity	High	Low
4/27/2020 3:34:55 PM	ps_user	Status	To investigate	To fix
4/27/2020 3:34:22 PM	jdoe	Comment	Triage of data race defects	Reassigning to project owner
4/27/2020 3:33:16 PM	jsmith	Severity	Unset	High
4/27/2020 3:33:10 PM	jsmith	Status	Unreviewed	To investigate
4/27/2020 3:33:06 PM	jsmith	Comment		Triage of data race defects

You can display changes for all the review fields, or you can filter for changes by **Status**, **Severity**, and **Comment**.

See Also


More About

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 26-2
- “Result Details in Polyspace Access Web Interface” on page 25-26

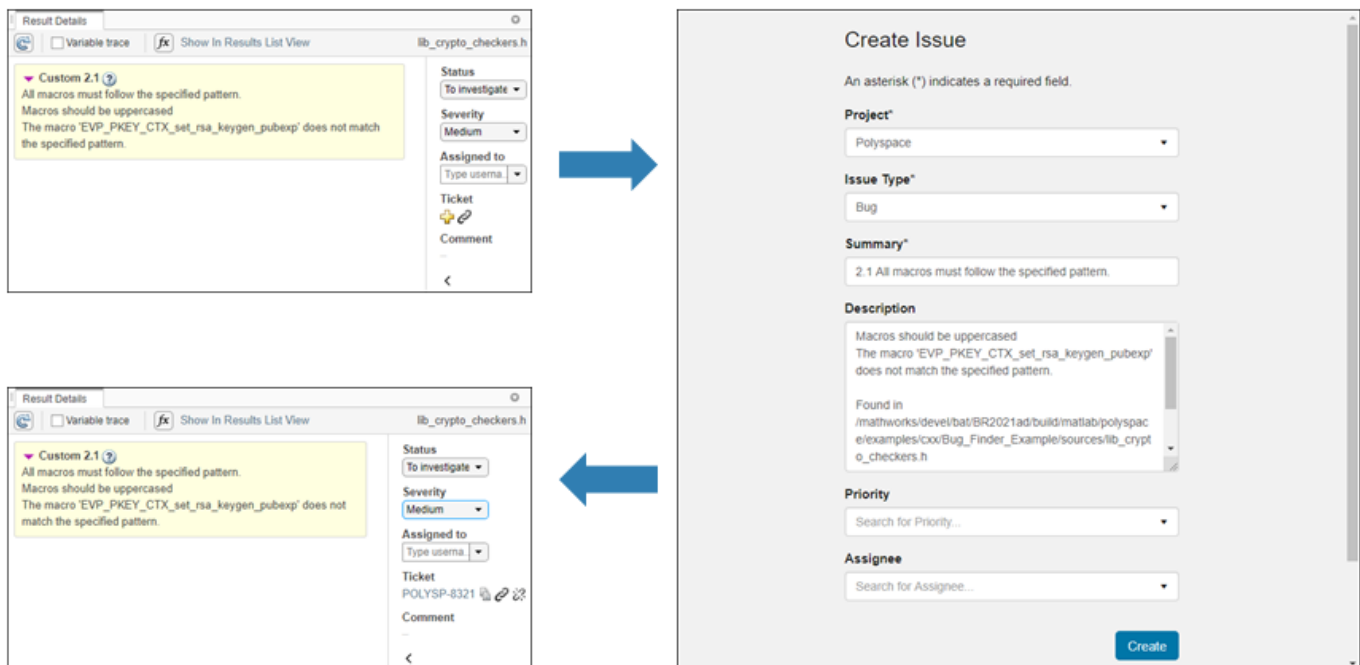
Create Bug Tracking Tool Tickets from the Polyspace Access Web Interface

If you use a bug tracking tool (BTT) such as Jira Software or Redmine as part of your software development process, you can configure Polyspace Access to create BTT tickets for Polyspace findings and add those tickets to the relevant project in your BTT software. See “Configure Issue Tracker”.

Create a Ticket

To create a BTT ticket, select one or more findings in the **Results list** and, from the **Results Details** pane, click  in Polyspace Access or **Create ticket** in the Polyspace desktop interface. To select multiple findings, press **CTRL** and click the findings.

Note In the desktop interface, you can create a BTT ticket only for results that you open from Polyspace Access.




If you use Jira, you may be prompted to enter your credentials. These credentials might be different from your Polyspace Access credentials.

After you create a BTT ticket, click the link in the **Results Details** pane to open the ticket in the BTT interface and track the progress in resolving the issue. For each finding that you selected when you created the ticket, the **Description** field of the ticket includes a URL to the Polyspace Access **Results List** filtered down to that finding.

Manage Existing Tickets

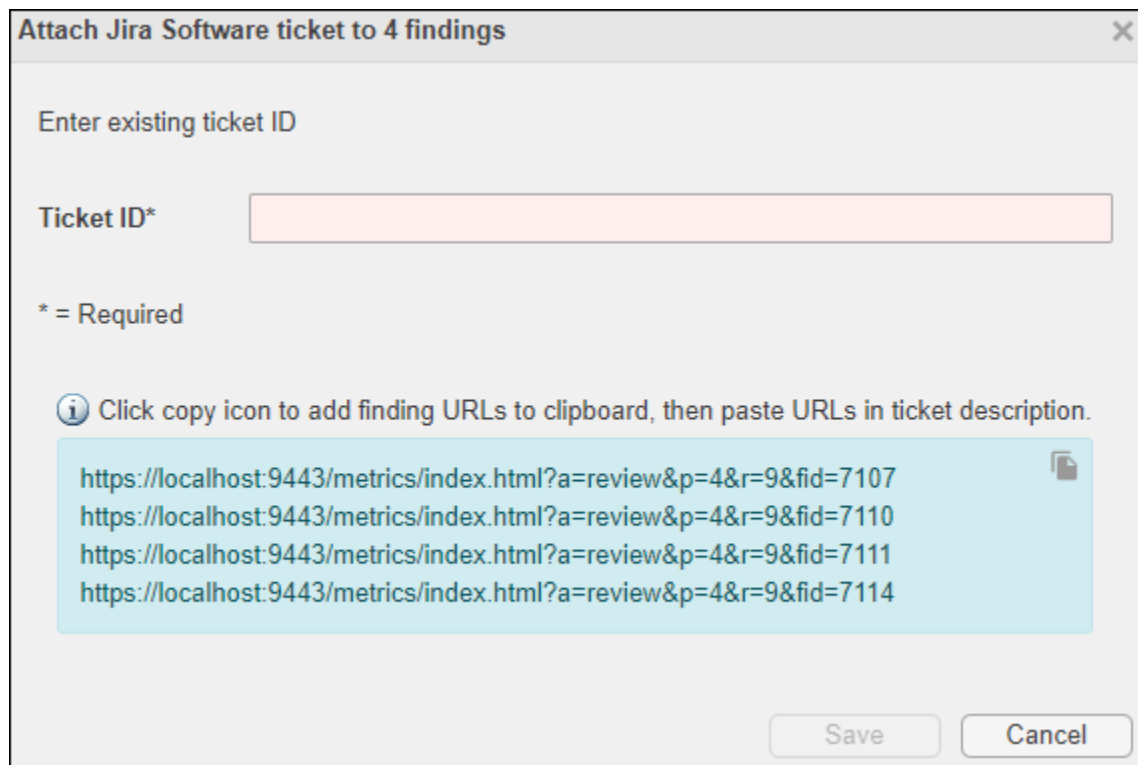
Once you create a BTT ticket, you can attach the ticket to additional findings or detach the ticket from findings associated with the ticket. To attach a ticket to additional findings:

- 1 Select findings in the **Results List** and then click  in the **Result Details**.
- 2 When prompted, enter the **ticket ID** in the dialogue window.

Click the copy icon in the **Result Details** pane of a finding already associated with the ticket to copy the **ticket ID**. The copy icon is not available when you select multiple findings with different ticket IDs. The **ticket ID** is also available in the **Ticket Key** column of the **Results List**.

- 3 Click the copy icon in the dialogue window to copy the findings URL, then click **Save**.
- 4 Click the ticket URL in the **Result Details** to open the ticket in the BTT interface and paste the findings URL you copied into the ticket description field.

You cannot attach more than one ticket to a finding. If a finding is already associated with a ticket, attaching a new ticket overwrites the existing **ticket ID**. This operation does not overwrite the ticket in your BTT. You can see all findings associated with a **ticket ID** by using the **Show only** text filter in the toolstrip.




Attach Jira Software ticket to 4 findings

Enter existing ticket ID


Ticket ID*

* = Required

 Click copy icon to add finding URLs to clipboard, then paste URLs in ticket description.

```
https://localhost:9443/metrics/index.html?a=review&p=4&r=9&fid=7107
https://localhost:9443/metrics/index.html?a=review&p=4&r=9&fid=7110
https://localhost:9443/metrics/index.html?a=review&p=4&r=9&fid=7111
https://localhost:9443/metrics/index.html?a=review&p=4&r=9&fid=7114
```

Save Cancel

To detach a ticket from a finding, select the finding in the **Results List**, then click  in the **Result Details**. The link to the ticket is removed from the **Result Details** pane. This operation does not remove the ticket in your BTT.

Note You cannot manage existing BTT tickets in the Polyspace desktop interface.

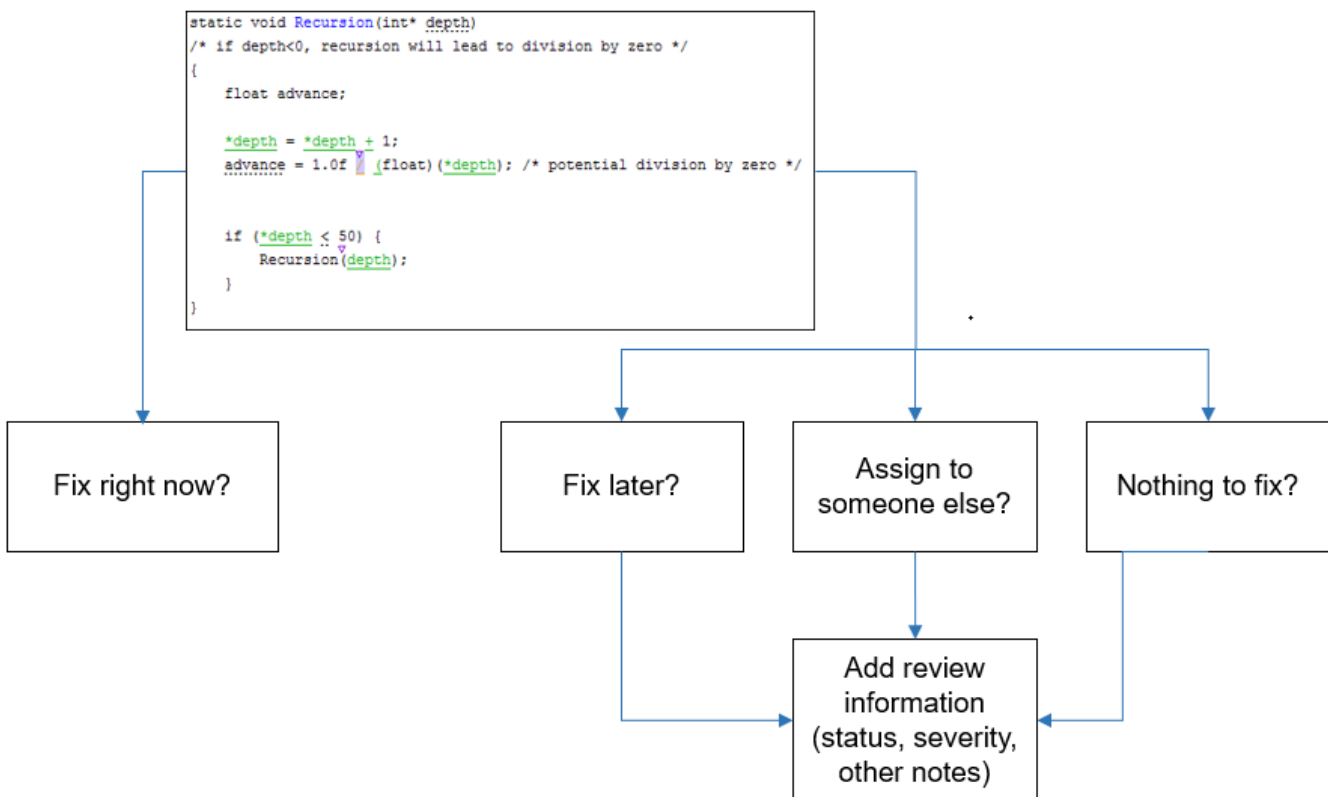
Fix or Comment Polyspace Results on Web Browser

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 26-2
- “Import Review Information from Existing Polyspace Access Projects” on page 26-5

Address Results in Polyspace Access Through Bug Fixes or Justifications

This topic describes how to add review information to Polyspace results in the Polyspace Access web interface. For a similar workflow in the user interface of the Polyspace desktop products, see “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 22-2.

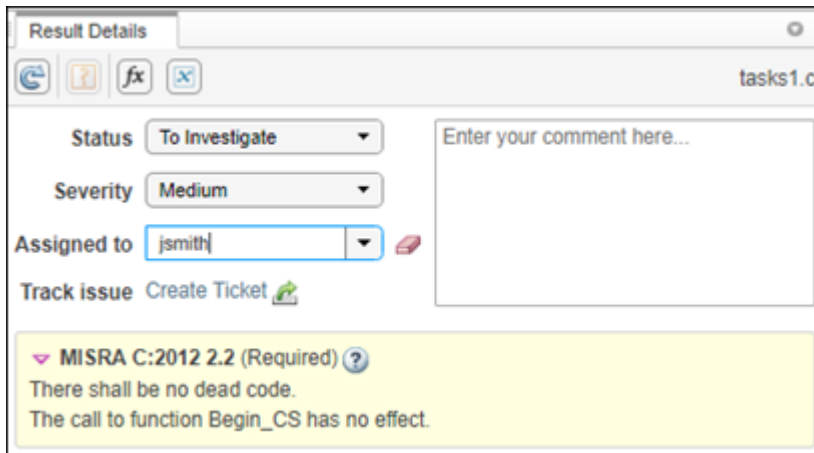
Once you understand the root cause of a Polyspace finding, you can fix your code. Otherwise, add review information to your Polyspace results to fix the code later or to justify the result. You can use the information to keep track of your review progress.



If you add review information to your results file, they carry over to the results of the next analysis on the same project. If you add the same information as comments to your code (annotate), they carry over to any subsequent analysis of the code, whether in the same project or not.

Add Review Information in Result Details pane

Set the **Status** and **Severity**, and optionally enter notes with more explanations in the **Result Details** pane. The status indicates your response to the Polyspace result.



If you do not plan to fix your code in response to a result, assign one of the following statuses:

- Justified
- No Action Planned
- Not a Defect

These statuses indicate that you have given due consideration and justified that result (retained the code despite the result). Note that subsequent analyses continue to show justified results as before. For instance, a Code Prover result that was previously orange does not turn green after justification. However, during review, you can filter out justified results in one click and focus only on results that are not justified. See “Filter and Sort Results in Polyspace Access Web Interface” on page 27-8.

Add Custom Status in Polyspace Access Project

If your company uses custom review statuses for results, you add those statuses in Polyspace Access. To add a custom **Status**:

- 1 Open the results in a Polyspace desktop interface. See “Open Polyspace Access Results in a Desktop Interface” on page 28-2.
- 2 In the Polyspace desktop interface, create a custom status. See “Create Custom Review Status” on page 2-24.

After you create the status, it is available from the **Status** dropdown in the **Results Details** pane. You can assign that custom status from Polyspace Access or from the Polyspace desktop interface.

Once you assign a custom status to a finding, you can apply that status to other findings in the same project. You cannot assign the custom status to findings in other projects.

Track Review Progress

To facilitate your review workflow in Polyspace Access, the findings are classified as:

- **To Do** — Findings with a status of Unreviewed that need to be addressed with a fix or a justification.
- **In Progress** — Findings with a status of To fix, To investigate, or Other that need to be addressed with a fix or a justification.
- **Done** — Findings with a status of Justified, No action planned, or Not a defect.

Note Green run-time checks, green shared variables, non-shared variables, and code metrics do not need to be addressed or justified. These findings do not count toward the number of findings that are **To Do, In Progress, and Done**.

In the **DASHBOARD** perspective, findings that are **To Do** or **In Progress** are considered as **Open Issues**. If a Polyspace analysis of your code finds known or acceptable defects or coding rule violations, you can remove the defects or violations from this list of **Open Issues** in subsequent analyses by assigning one of the justified statuses outlined above.

Comment or Annotate in Code

You can also add specific code comments or annotations in a code editor in response to Polyspace results. If you enter code comments or annotations in a specific syntax, on the next analysis of the code, the software can read them and populate the **Severity, Status, and Comment** fields in the result details.

For the annotation syntax, see “Annotate Code and Hide Known or Acceptable Results” on page 30-2.

If you do not explicitly specify a status in your annotation, Polyspace assumes that you have set a status of **No Action Planned**.

See Also

More About

- “Annotate Code and Hide Known or Acceptable Results” on page 30-2

Import Review Information from Existing Polyspace Access Projects

This topic describes how to import review information from previous results in Polyspace Access. For information on importing from results that are not uploaded to Polyspace Access, see “Import Review Information from Previous Polyspace Analysis” on page 20-2.

If you review findings in a Polyspace Access project and you reuse the same source code that contains those findings in another project, you can import the existing review information to the other project. You do not need to review the findings again in the other project. The project you import from is the source project. The project you import to is the target project.

For instance, suppose your team has reviewed all findings for the file `customClass.cpp` in the Polyspace Access project `components/oldProject (BF)`. If you use `customClass.cpp` in a different project, you can import review information such as **Status** and **Severity** from `components/oldProject (BF)` into the other project.

Note that when you upload a run to a project, Polyspace Access automatically imports the review information from previous runs of that project to the newly uploaded run.

These values are imported when you merge review information between projects:

- **Status**
- **Severity**
- **Assigned to**
- **Comments**
- **Ticket Key**


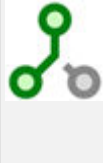
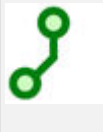
Import Review Information from Source Project to Target Project in Polyspace Access

Import review information from a source project to a target project.

- 1 In the **Project Explorer**, right-click your target project and select **Import Reviews from Another Project**.
- 2 Locate and select the source project. You can only import from one source at a time.
- 3 Select one of these import policies:
 - **Write review when there is no review on target finding**
Import non-empty source project review information only if all the review fields of the target finding are empty.
 - **Source reviews always replace target**
Import non-empty source project review information and replace target fields, even if the target review already contains review information.
- 4 Click **OK**. A dialog box appears at the top of the **Dashboard** when the import begins and when it completes. Imports for larger projects can take several minutes to complete.

View and Select Imported Reviews

Click **Open Results** in the **Project Overview** dashboard to view the **Results List**. Project results with imported review information have an icon next to them in the **Family** column. The icon indicates the current state of the imported review. This table describes the states of imported reviews.

Icon	Status	Status Description
	Not Applied	Review information from the source project is not applied to the target project findings.
	Overwritten	Review information from the source project is applied to the target project finding. The source overwrites the original target review information.
	Written	Review information from the source project is applied to previously empty target project findings.

You can switch between original and imported review information. To decide what review information to use, view imported and original result information side-by-side. Use the **Imported Review Selection** window to view result information in this way. To access this pane:

- Right-click the result you want to review in the **Results List**.
- Select **Show Imported Review Selection**.

In the **Imported Review Selection** panel, three columns represent the **Review Fields**, the **Original Values**, and the **Imported Values**. A radio button next to **Original Values** and **Imported Values** enables you to quickly select which values to apply to the findings.

Confirm Imported Review Information

Imported review information is considered unconfirmed until you manually confirm it. You can confirm result information individually or as a group.


- 1 Expand the **Filters** list and select the **Unconfirmed Imports** filter.



- 2 Select a result. To select multiple results, click those results while holding the **Ctrl** key. To select a range of adjacent results, click the first and last result in that range while holding the **Shift** key.
- 3 Right-click the selected result, go to **Confirm imported reviews** in the context menu and choose one of these options.

Option	Option Description
Use original values	Keep the original review information in the target finding for all selected results. If this option is grayed out, the target finding already uses the original review information.
Use imported values	Apply the imported review information to the target finding for all selected results. If this option is grayed out, the target finding already uses the imported review information.
Confirm current selection	Confirm review information as it currently is set. If this option is grayed out, the target finding is already confirmed.

In some instances, it is useful to know the review history of a result. To open the **Review History**

pane, from the **Result Details** pane, click the  icon. The **Review History** shows information about changes to individual result details including the name of the editor and time of the edit. See “Review History in Polyspace Access Web Interface” on page 25-33.

Import Review Information at the Command-Line

To import review information from an existing project to another project that reuses the code that contains the reviewed finding, use command `polyspace-access -import-reviews` and specify:

- The full path of the project that you want to import the review information from (source project).
- The full path of the project that you want to import the review information to (target project).
- Optionally, you can specify one of these import strategies:
 - `never-overwrite-target` (default) — If a review field in the target project is already filled, do not overwrite it with the content from the source project
 - `always-overwrite-target` — Always overwrite the content of the review fields in the target project, even if they are already filled.

For example, if you have already reviewed findings in project `public/example/branchA`, and you reuse the reviewed code in project `public/example/branchB`, use this command to import the review information from `branchA` to `branchB`:

```
polyspace-access $login -import-reviews public/example/branchA \
-to-project-path public/example/branchB
```

Here, `$login` is a variable that stores the login credentials and other connection information. To configure this variable, see “Encrypt Password and Store Login Options in a Variable”.

After you complete the import, you might want to examine the result of the import operation. Use the `polyspace-access -export` command with option `-imported-reviews` and filter by one of these values:

- `Not applied` — Review information was imported from the source project but the review fields in the target project kept the original values.

- **Written** — The Review information from the source project was written to the target project only if the review fields in the target project were previously empty.
- **Overwritten** — The Review information from the source project was written to the target project even if the review fields in the target project were not previously empty.
- **Unconfirmed** — Use this filter to select findings that where the result of the import operation has not been confirmed by a reviewer. You confirm the result of the import operation in the Polyspace Access interface. See “Confirm Imported Review Information” on page 26-6 .

For example, to get a list of findings from the target project whose review information was overwritten, enter this command:

```
polyspace-access $login -export public/example/branchB \  
-imported-reviews Overwritten -output overWrittenReviews.txt
```

The command output tab-separated values (TSV) file `overWrittenReviews.txt` which contains only findings that had their review information overwritten in the target project.

See Also

`polyspace-access | polyspace-bug-finder-server`

Related Examples

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 26-2

Manage Results

- “Manage Permissions and View Project Trends in Polyspace Access Web Interface” on page 27-2
- “Filter and Sort Results in Polyspace Access Web Interface” on page 27-8
- “Create Custom Filter Groups in Polyspace Access Web Interface” on page 27-11
- “Manage Software Quality Objectives in Polyspace Access” on page 27-13
- “Add Labels to Project Runs in Polyspace Access” on page 27-16
- “Compare Results in Polyspace Access Project to Previous Runs and View Trends” on page 27-19

Manage Permissions and View Project Trends in Polyspace Access Web Interface

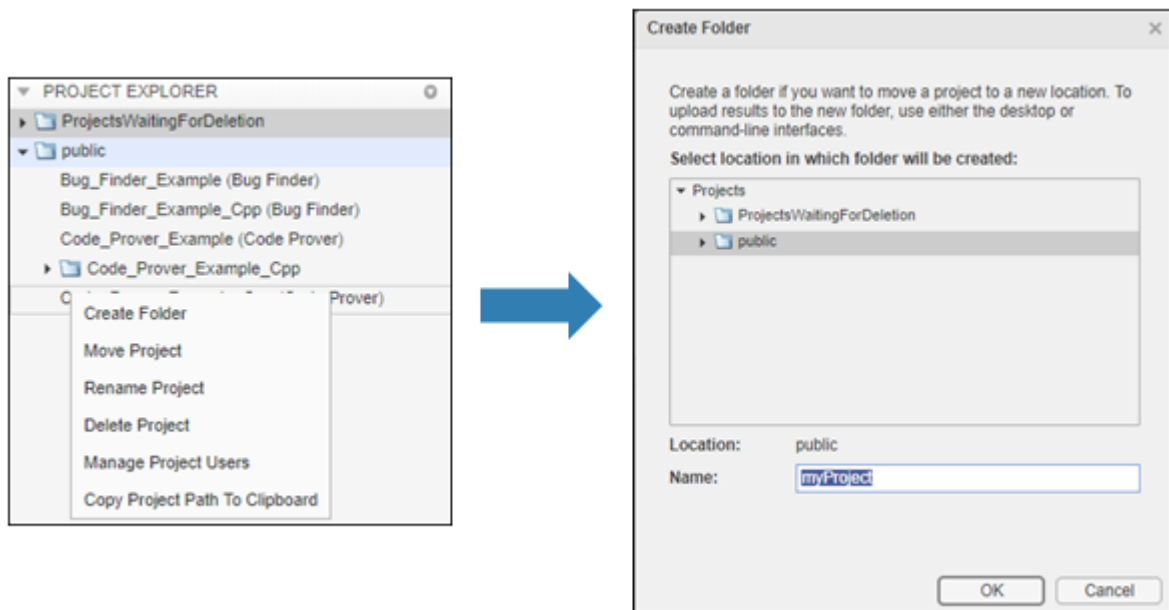
Before you start reviewing the overall quality of a project and investigating findings in your code, create project folders and set permissions to allow or restrict team members access to your projects.

Create a Project Folder

To facilitate the review process, create folders in Polyspace Access to group related results.

Create Folder from the Polyspace Access Interface

From the **Project Explorer** in the **DASHBOARD** perspective, select any existing folder or project and click **Create Folder** in the context menu. In the **Create Folder** window, click an existing folder to create a subfolder. To create a folder at the top of the **Project Explorer** hierarchy, click **Projects**.



Create Project Folder at Command Line

To create a folder in Polyspace Access from the DOS or UNIX command lines, use the `polyspace-access` binary. This binary is available under the `polyspaceroot/polyspace/bin` folder. The `polyspaceroot` folder is the Polyspace product installation folder, for example `C:\Program Files\Polyspace Server\R2023a`.

For instance, to create `myProject` under the folder `myRelease`, use this command:

```
polyspace-access -host hostName -port port -create-project myRelease/myProject
```

`hostName` and `port` correspond to the host name and port number that you specify in the URL of the Polyspace Access interface, for example `https://hostName:port/metrics/index.html`. If you are unsure about which host name and port number to use, contact your Polyspace Access

administrator. Depending on your configuration, you might also need to specify the `-protocol` option in the command.

Manage Project Permissions

To set permissions for folders or projects in Polyspace Access, assign roles to users or groups. The permissions that correspond to each role are listed in this table.

Role	Permission
Administrator	<ul style="list-style-type: none"> Move, rename, or delete any folder or project and review their content. Assign roles Owner, Contributor, or Forbidden to other users or groups. View and manage contents of ProjectsWaitingForDeletion folder. See “Delete Project Runs or Entire Projects”. <p>To set a user as Administrator, see “Configure User Manager”.</p> <p>You cannot move a folder or project to a new location if a folder or project with the same name already exists at that location.</p>
Owner	<ul style="list-style-type: none"> Move, rename, or delete folders or projects that you own and review their content. Assign roles Contributor or Forbidden to other users or groups. <p>You are the owner of folders that you create and of project results that you upload.</p> <p>You cannot move a folder or project to a new location if a folder or project with the same name already exists at that location.</p>
Contributor	<ul style="list-style-type: none"> Review content of folders or projects for which you are a contributor. See the roles of other users and groups for the project.
Forbidden	<p>No access to the specified folder or project. Set this role to restrict the access of a user or group if:</p> <ul style="list-style-type: none"> The user or group inherits access from a parent folder. The user or group inherits access from a parent group.

- The user or group roles that you assign for a project folder apply to all the projects and subfolders under that folder. You can also set different user or group roles for each project or subfolder. For instance, you can assign user `jsmith` as a contributor for folder `myRelease`, and then restrict the access of `jsmith` to subfolder `myRelease/update1`.
- Only **Administrator** or **Owner** roles can allow or restrict the access of other team members or groups to a project or folder.
- Only **Administrator** roles can assign other users or groups as owners of a project or folder.
- Unless you explicitly set a user or group role for a project, the user or group inherit the role of their parent group for that project. For instance, if user `jsmith` is not assigned any role for folder `myRelease`, and `jsmith` is a member of a group that is a contributor for folder `myRelease`, then `jsmith` is also a contributor to folder `myRelease`.

By default, all users are members of the **Polyspace Access public group** and all users inherit the role of that group (**Contributor**) for the **public** folder. You cannot change the permissions for the **public** folder, but you can change permissions for subfolders or projects inside the **public** folder.

Manage Permissions in Polyspace Access Web Interface

From the **Project Explorer** in the **DASHBOARD** perspective, select any existing folder or project and click **Manage Project Permissions** in the context menu.

The **Manage Project Permissions** interface opens for the selected project.



- To assign or unassign roles, right-click a user or group in any of the panes.
- Place your cursor over a user or group in any of the panes to see a tooltip that has information about the user or group role for the selected project, and whether that role is inherited from a parent project or parent group.

The screenshot shows the 'Manage Project Permissions' dialog box. The title bar reads 'Manage Project Permissions'. The interface is divided into several sections:

- User Search:** A search bar contains 'ja'. Below it, a 'Search Results' list shows Jane Smith, Jared Dunn, and James Developer.
- Selection Details:** A list shows 'Dev Managers', 'Group backend API', and 'Polyspace Access public group'.
- Administrator Pane:** Labeled 'Interface refactoring', it lists 'Administrator' with a sub-entry 'admin'.
- Owner Pane:** Labeled 'Owner', it lists 'Ada Lovelace'.
- Contributor Pane:** Labeled 'Contributor', it lists 'Group backend API', 'Jane Smith', and 'James Developer'.
- Forbidden Pane:** Labeled 'Forbidden', it lists 'Richard Roll'.

A 'Close' button is located at the bottom right of the dialog.

This table provides additional information about the different panes in the **Manage Project Permissions** interface.

Pane	Description
Search Results	To view a list of user or groups that match your search string, type the user name or group name in the search bar.
Selection Details	From any of the other panes, click a user to view the groups that the user belongs to in this pane. If you click a group, this pane shows only the direct descendant members of the group. For instance, if group <code>nestedGroup</code> is a member of group <code>parentGroup</code> , when you click <code>parentGroup</code> , you do not see the members of <code>nestedGroup</code> in this pane.
Administrator Owner Contributor Forbidden	<ul style="list-style-type: none"> View list of users or groups that have a role assigned for the project. If the role is assigned explicitly on the selected project, the user or group icon is black, for instance . If the role is inherited from a parent folder, the user or group icon is gray, for instance . These four panes do not show users or groups that inherit their role from a parent group. For instance, if group <code>Contractors</code> is assigned as a contributor to the project, the members of this group are not listed in the Contributor pane.

The list of Polyspace Access users and groups (identities) is populated from the **User Manager** database. If an identity is removed from this database and the identity was assigned a role explicitly on at least one Polyspace Access project, that identity is highlighted in red in the **Manage Project Permissions** interface and is listed by ID instead of display name, for instance `jsmith`, instead of John Smith. A role is not explicitly assigned if it is inherited from a parent group or a parent project folder.

Contact a Polyspace administrator to remove that identity from the Polyspace Access. See “Update List of Polyspace Access Users and Groups”.

Identities that are deleted from the **User Manager** database and that do not have roles explicitly assigned to them are removed from Polyspace Access when you refresh your web browser.

Manage Permissions at Command Line

To manage access to uploaded results from the DOS or UNIX command lines, use the `polyspace-access` binary. This binary is available under the `polyspaceroot/polyspace/bin` folder. The `polyspaceroot` folder is the Polyspace product installation folder, for example `C:\Program Files\Polyspace Server\R2023a`.

For instance, to assign `jsmith` as **Contributor** for project `myProject`, use this command:

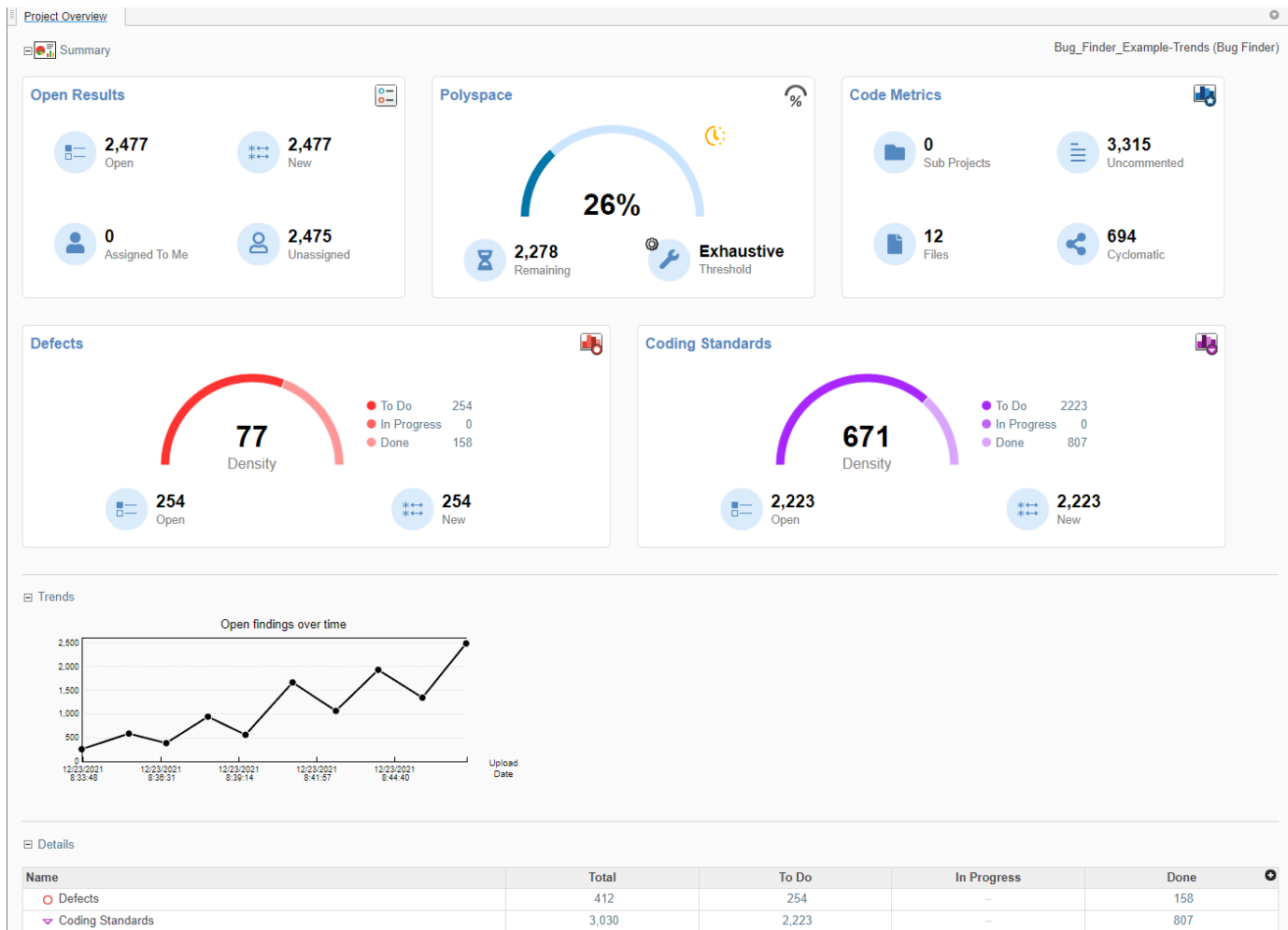
```
polyspace-access -host hostName ^
-set-role contributor -user jsmith ^
-project-path myProject
```

hostName and *port* correspond to the host name and port number that you specify in the URL of the Polyspace Access interface, for example `https://hostName:port/metrics/index.html`. If you

are unsure about which host name and port number to use, contact your Polyspace Access administrator. Depending on your configuration, you might also need to specify the `-protocol` option in the command.

You cannot assign the **Administrator** role to a user at the command line.

View Project Trends



In the **DASHBOARD** perspective, select the project that you want to investigate from the **PROJECT BROWSER**.

If you select a folder that includes multiple projects, the dashboard displays an aggregate of results for all the projects that you have permission to view. If the folder contains a project for which you are not an **Administrator**, **Owner**, or **Contributor**, results for that project are not included in the aggregate calculation.

In the **Project Overview** dashboard, you see a summary of **Open Issues**, including the number of **New** results since the previous analysis run and the number of results that are **Unassigned**.

Other cards provide statistics for each family of findings. The **Run-time Checks** card shows the **Selectivity**, that is, the percentage of all findings that are green. When you enable the calculation of

code metrics in your analysis, the **Defects** and **Coding Standards** cards show the **Density**, the number of findings per thousand lines of code without comments.

In the **Details** section, you see the review progress for each family of results. The results are classified as:

- **To Do** — Findings with a status of **Unreviewed** that need to be addressed with a fix or a justification.
- **In Progress** — Findings with a status of **To fix**, **To investigate**, or **Other** that need to be addressed with a fix or a justification.
- **Done** — Findings with a status of **Justified**, **No action planned**, or **Not a defect**.

Note Green run-time checks, green shared variables, non-shared variables, and code metrics do not need to be addressed or justified. These findings do not count toward the number of findings that are **To Do**, **In Progress**, and **Done**.

If the number of open issues increases, open additional dashboards by using the buttons in the **DASHBOARDS** section of the toolbar. Each button opens a dashboard for a family of findings, for instance **Defects**. To determine the root cause of the increase, use the information on these dashboards. Once you determine the set of findings that you want your team to focus on, open the **REVIEW** perspective to start managing the results. See “Manage Results”.

See Also

More About

- “Upload Results to Polyspace Access” on page 2-25

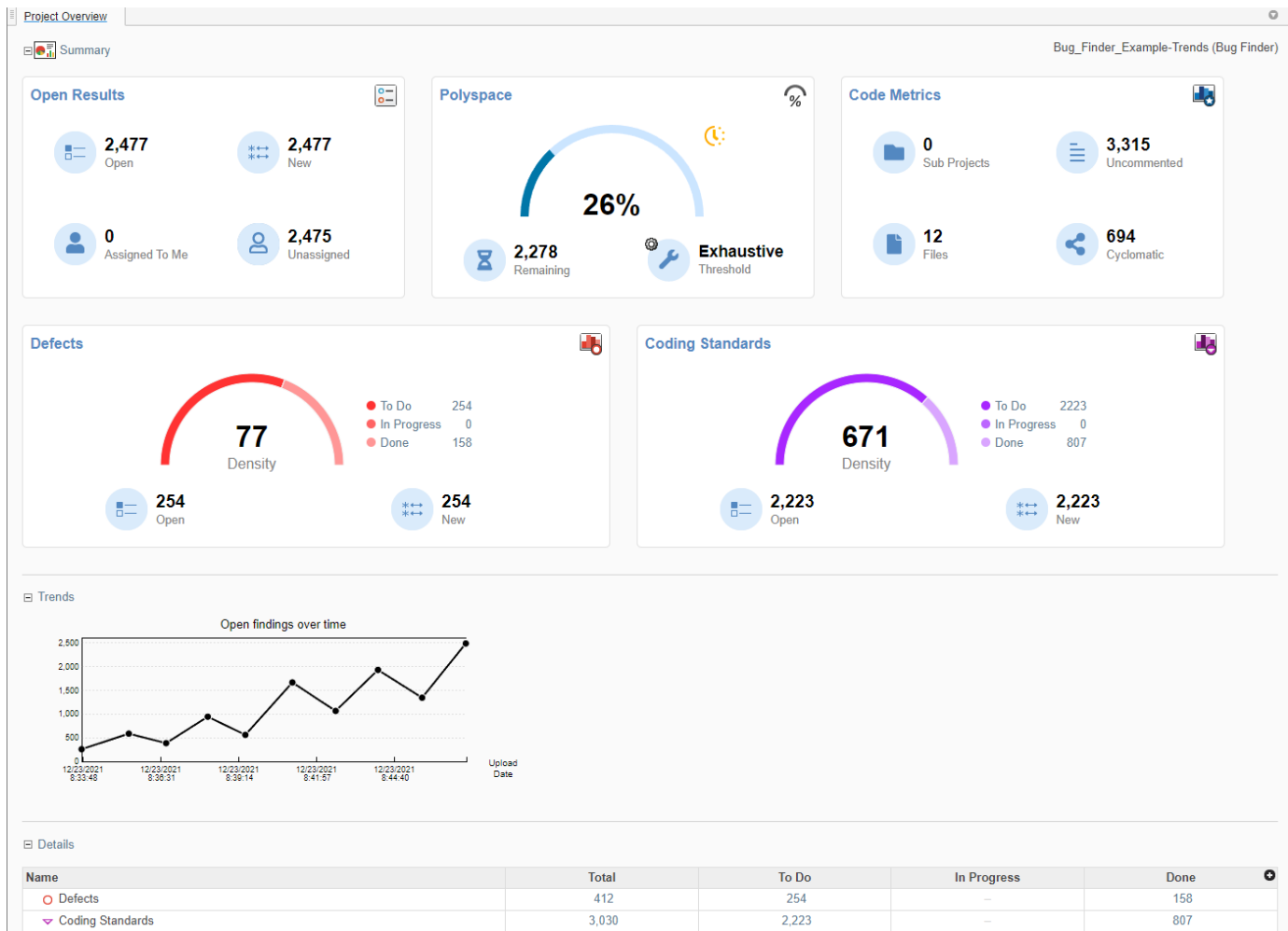
Filter and Sort Results in Polyspace Access Web Interface

This topic describes how to filter, sort, and otherwise manage results in the Polyspace Access web interface. For a similar workflow in the user interface of the Polyspace desktop products, see “Filter and Group Results in Polyspace Desktop User Interface” on page 23-2.

When you open the results of a Polyspace analysis in the **DASHBOARD** view of Polyspace Access, you see statistics about your project in the **Project Overview** dashboard. The statistics cover findings for:

- Bug Finder “Defects”.
- Code Prover “Run-Time Checks” (Polyspace Code Prover).
- “Coding Standards” violations.
- “Code Metrics” and “Evaluate Polyspace Bug Finder Results Against Bug Finder Quality Objectives” on page 31-2 compliance.

To organize your review, you can narrow down the list or group results by file or result type.



Some of the ways you can use filtering are:

- You can display only certain types of defects or run-time checks.

For instance, for a Bug Finder analysis, you can display only high-impact defects. See “Classification of Defects by Impact” on page 18-49.

- You can display only new results found since the last analysis or since a previous analysis. See “Compare Results in Polyspace Access Project to Previous Runs and View Trends” on page 27-19.
- You can display only the results that you have not justified. Results that are not justified are considered **Open**. They are results with status Unreviewed, To Investigate, To Fix, or Other.

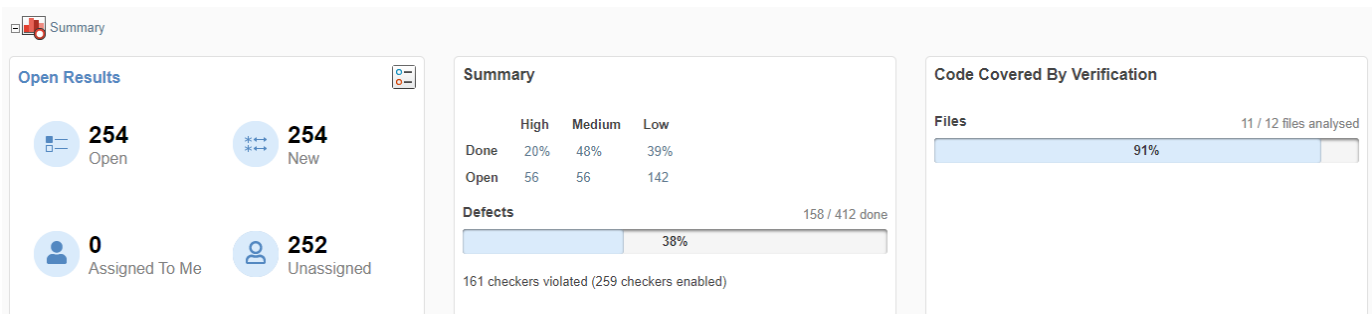
For information on justification, see “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 26-2.

- You can display only results that you still need to address to reach a **Quality Objectives** threshold.

Filter Results

You can filter results by drilling down on a set of results in a dashboard, or directly in the **Results List** pane by using the **REVIEW** toolstrip filters.

Filter Using Dashboards

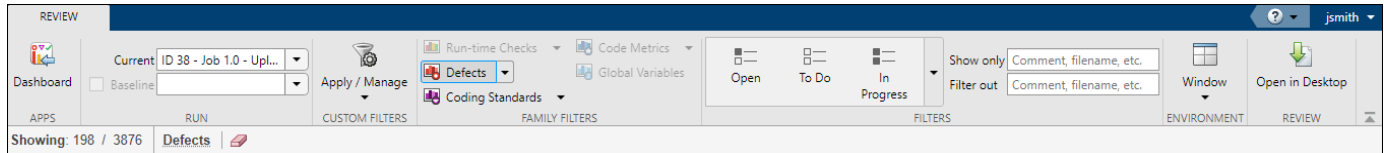


In the **DASHBOARD** view, you can:

- Click a section of a pie chart or a pie chart legend on the **Project Overview** dashboard to see the corresponding set of results.
- Open dashboards for different families of results, then click a number to open a list filtered to the corresponding set of results. For instance:
 - To see only high-impact defects that are still **Open** in a Bug Finder analysis, click the corresponding number in the **Summary** section of the **Defects** dashboard. **Open** results have status Unreviewed, To Investigate, To Fix, or Other.
 - To see only red checks that are **Done** in a Code Prover analysis, click the corresponding number in the **Summary** section of the **Run-time Checks** dashboard. **Done** results have status Justified, No Action Planned, or Not A Defect.
 - To see violations of the MISRAC C:2012 coding standards in a particular file, use the table in the **Details** section of the **MISRA C:2012** dashboard.
- Compare the **Current** run to an earlier **Baseline** run and review **New** or **Unresolved** findings. See “Compare Results in Polyspace Access Project to Previous Runs and View Trends” on page 27-19.

If you select a folder that contains multiple projects in the **Project Explorer**, the dashboards display an aggregate of results for all the projects. Most of the fields in the dashboard are not clickable when you look at the statistics for multiple projects.

Filter Using REVIEW Toolstrip



In the **REVIEW** view, you can filter results by families of Polyspace results (**FAMILY FILTERS**), or by result review progress (**FILTERS**). For instance:

- To see Bug Finder defects only, select the **Defects** filter in the **FAMILY FILTERS** group.
- To see only results that are not justified, select the **Open** filter in the **FILTERS** group.

The filter bar underneath the toolstrip shows how many findings are displayed out of the total findings, along with which filters are currently applied.

Note If you are reviewing a filtered list of open results and you add an SQO filter, the number of filtered results might increase. This can happen when your project has code metrics with a status of FAIL. The SQO filter adds the failing code metric results to the list of results.

The buttons in the **FILTERS** section of the toolstrip are global. They apply to all families of findings.

To filter results by specific content, such as a function name, use the **Show only** or **Filter out** text filters. These filters match the text you enter against the content of all the columns in the “Results List in Polyspace Access Web Interface” on page 25-19. For instance, if you enter `foo` in the **Filter out** filter, the **Results List** hides all the results that contain `foo` in any of the **Results List** columns.

You can also filter results by right-clicking the content of a column in the **Results List**. This action is equivalent to entering the content directly in the **Show only** or **Filter out** text filters. For instance, if you right-click `foo` in the **Function** column, the filter applies to all results that contain `foo` in any of the **Results List** columns.

Filters you apply do not carry over to the next analysis.

See Also

More About

- “Classification of Defects by Impact” on page 18-49

Create Custom Filter Groups in Polyspace Access Web Interface

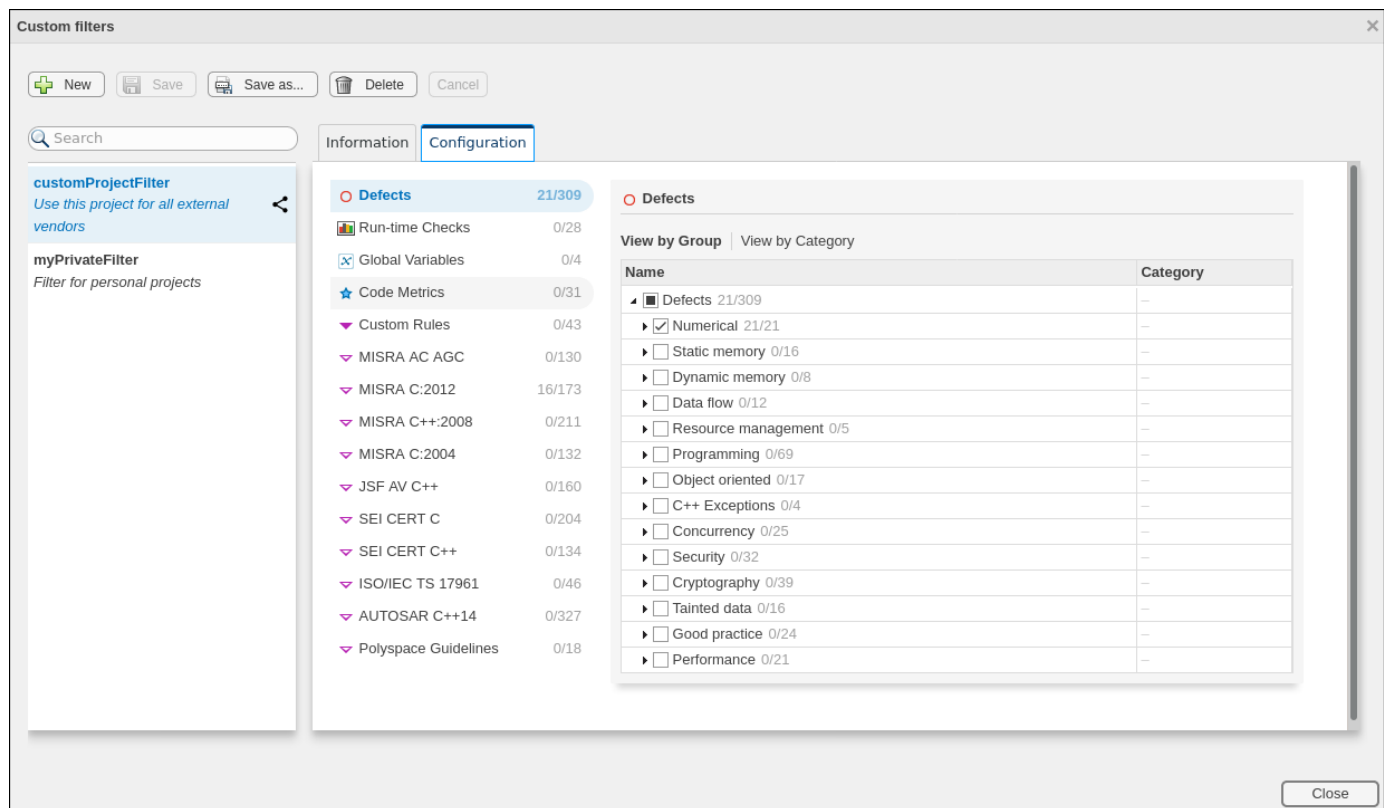
When you review results in the **Results List**, you can apply filters from the **FAMILY FILTERS** section of the toolstrip to focus your review on specific Polyspace families of results, such as:

- Bug Finder “Defects”.
- Code Prover “Run-Time Checks” (Polyspace Code Prover) and “Global Variables” (Polyspace Code Prover).
- “Coding Standards” violations.
- “Code Metrics”.

Define custom filters to narrow the scope of your review to only findings that are relevant to your project or organization. For instance, you might be interested in reviewing only **Numerical** Bug Finder defects and violations of **Mandatory MISRA C:2012** rules.

Once you define custom filters, you can share those filters with other Polyspace Access users to ensure consistent review scopes across your projects or organization.

To create or edit a custom filter, click **Apply/Manage > Manage filters**.



To create a new filter, in the **Custom filters** window, click **New** and then enter the filter name in the **New Custom Filter** pop-up window. You can optionally provide a description and enable the **Shared filter** checkbox to share the filter with other Polyspace Access users.

By default, custom filters are private and can be viewed only by the user who creates the filter. A private filter can be edited only by the user who creates that filter. A shared filter can be edited by the user who creates the filter or by a user with the role of **Administrator**.

To make changes to a filter name, description, or to enable or disable filter sharing, go to the **Information** tab.

To edit the filter selection, on the **Configuration** tab, click a Polyspace results family, for instance MISRA C:2012, and then select a node or expand the node to select individual results. For each family of results, you can view the nodes by group or by category when available.

To save your changes, click **Save** or **Save as** to save your edits in as new custom filter.

Apply custom filters by selecting the appropriate filter from **Apply/Manage > Private filters** or **Apply/Manage > Shared filters**. You can apply more than one custom filter, including combinations of private and shared filters.

Custom filters do not apply to the **DASHBOARD** view.

See Also

Related Examples

- “Filter Results” on page 27-9
- “Customize Software Quality Objectives” on page 25-16

Manage Software Quality Objectives in Polyspace Access

To monitor the quality of your code against predefined on page 31-2 software quality thresholds or user-defined thresholds, use the **Quality Objectives** dashboard. See “Quality Objectives Dashboard in Polyspace Access” on page 25-14.

The first time that you upload results to a new project, Polyspace Access assigns the default **Polyspace Software Quality Objectives** definition to that project. To create custom software quality objective (SQO) definitions, see “Customize Software Quality Objectives” on page 25-16.

You can manage the SQO of a project from the user interface or at the command line.

After you assign an SQO definition, you see the label  (not computed) on the **Quality Objectives** card and dashboard until the project statistics are recalculated.

The SQO statistics for a project are recalculated when:


- You upload a new run for the project.
- You select a finding and make a change to any of the fields in the **Result Details** pane.

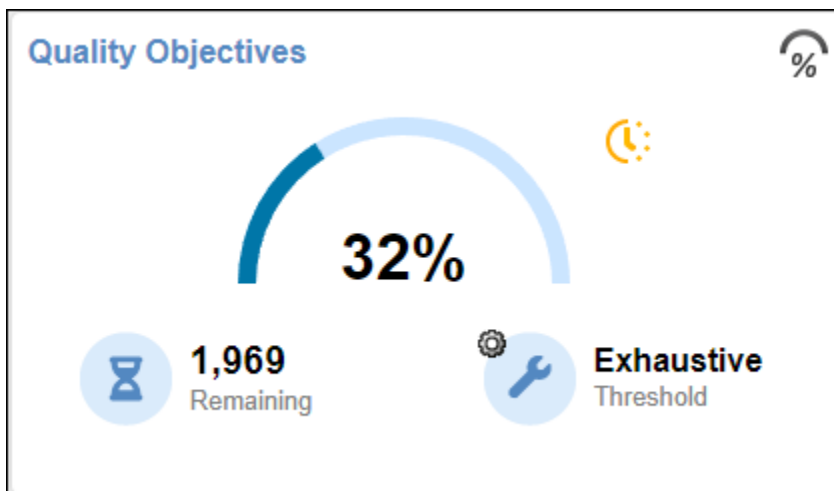
The SQO definition that you assign to a project applies only to runs that you upload to the project after assigning the definition.

Tip When the **Quality Objectives** settings and the calculated statistics for a project are out of sync, the **Quality Objectives** dashboard displays a warning .

If you delete an SQO definition, Polyspace Access assigns the **Polyspace Software Quality Objectives** to all the projects to which the deleted definition was assigned.

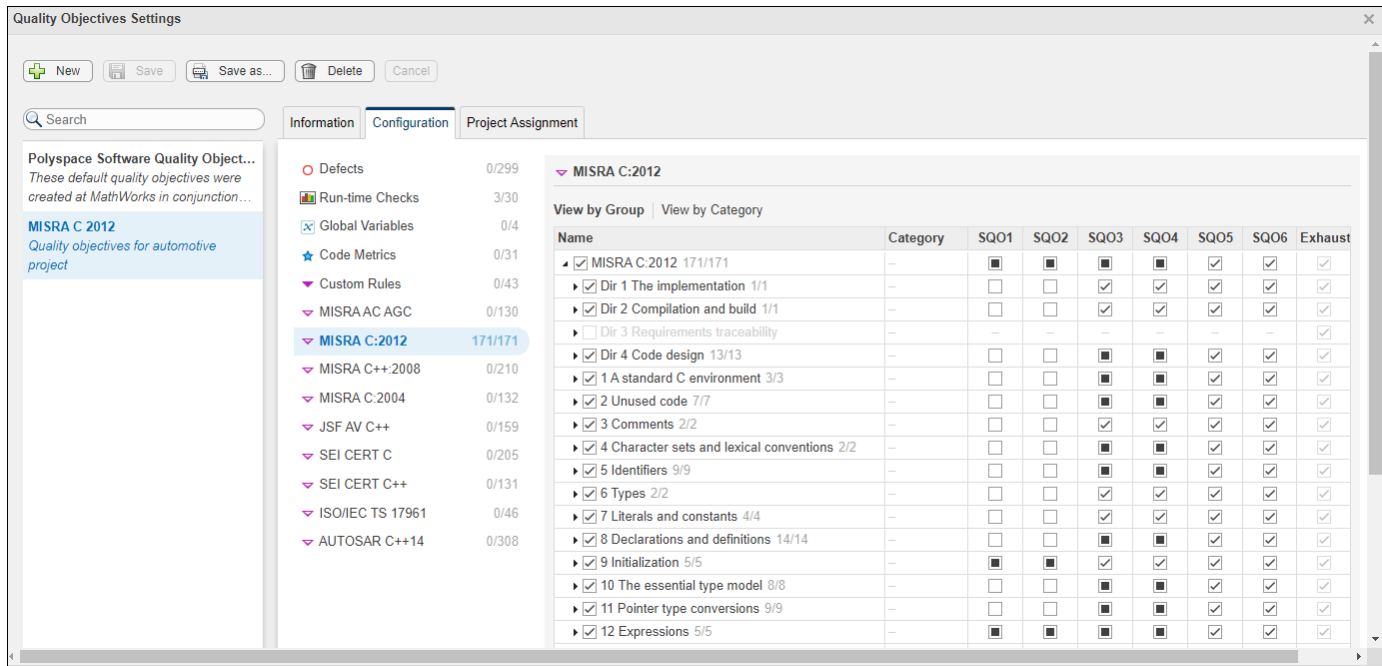
Manage SQOs in the User Interface

To assign an SQO definition or level to a project, right-click a project in the **Project Explorer** or click  on the **Quality Objectives** card or the **Quality Objectives** dashboard.



If you make changes to a quality objectives definition that applies to multiple projects, Polyspace Access displays a warning with a link to the **Project Assignment** tab on the **Quality Objectives Settings** window. Open the tab to determine which projects are affected by your changes and inform users that are contributors to those projects of the changes to the quality objectives definition.

To view which projects an SQO definition is assigned to, go to the **Project Assignment** tab in the **Quality Objectives Settings**.



Manage SQOs at the Command Line

To manage SQOs from the command line, use the `polyspace-access` command. In the following examples, `$LOGIN` is a variable that stores the login credentials and other connection information. To configure this variable, see “Encrypt Password and Store Login Options in a Variable”.

You can:

- Assign an SQO level, and optionally an SQO definition to a project. For instance, to assign level 3 of SQO definition `My Custom SQO` to project `myProject` with project path `public/examples/myProject`, enter this command:

```
polyspace-access -set-sqo public/examples/myProject -level 3 -name "My Custom SQO" $LOGIN
```

Option `-level` is mandatory and can be any value from 1 to 6 or "exhaustive", while option `-name` is optional.

If you do not use `-name`, the level that you specify is applied for the currently assigned SQO definition.

- View the currently assigned SQO definition and SQO level for a project. For instance, to view the assigned SQO level and definition for project `myProject` with project path `public/examples/myProject`, enter this command:

```
polyspace-access -get-sqo public/examples/myProject $LOGIN
```

The command outputs the SQO name and level in this format:

Current Quality Objectives: NAME My Custom SQO LEVEL SQO-3

- View a list of all currently available SQO definitions. Enter this command:

```
polyspace-access -list-sqo $LOGIN
```

See Also

More About

- “Quality Objectives Dashboard in Polyspace Access” on page 25-14
- “Evaluate Polyspace Bug Finder Results Against Bug Finder Quality Objectives” on page 31-2
- “Code Metrics”

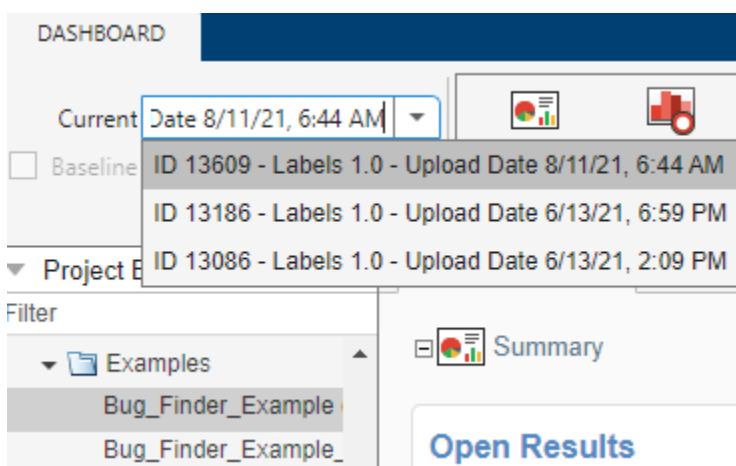
Add Labels to Project Runs in Polyspace Access


To help identify project runs uploaded to Polyspace Access, you can assign custom labels to runs. Custom labels are in addition to the unique run IDs that Polyspace Access assigns to each run.

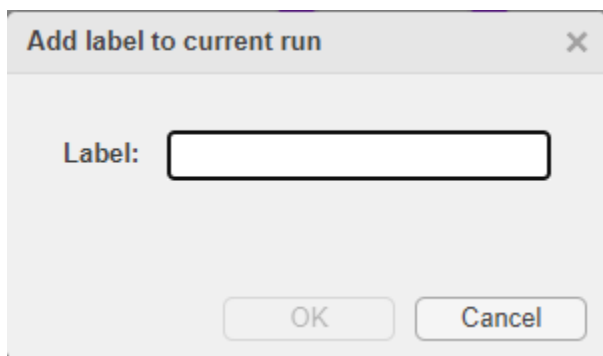
You can assign labels from the user interface through the **Project Details** pane on the **Dashboard**, or from the command line.

Manage Labels in the User Interface

To add a label to a run, first select a project in the **Project Explorer**. Select the run that you want to add a label to by using the **Current** drop-down list.



In the **Project Details** pane, in the **Labels** box under the **Run** section, click the  icon. In the **Add label to current run** box, enter the label name to assign to the run, and then click **OK**.



Labels are sorted in alphabetical order. There is no limit on how many labels you can assign to a single run.

To delete a label, select the label and click the  icon. You can select multiple labels to delete them simultaneously.

▼ Project Details

Project

Name Bug_Finder_Example (Bug Finder)

Author MathWorks

Language C

Tools Bug Finder

Coding Standards Custom Rules, Guidelines



Number of Runs 1

Run (ID 6545)

Upload Date 11/25/20, 10:21 AM

Labels

- 1
- example
- test

Manage Labels at the Command Line

To add a label to a run from the command line, use the `polyspace-access` command with the `-add-label` option.

For instance, suppose that you use a continuous integration tool like Jenkins and that you want to associate the Jenkins build number with the project run that you upload to Polyspace Access. The following steps show how to extract the run ID of the uploaded project run and add a label to that run by using Bash commands:

- 1 Store the output of the `polyspace-access -upload` command to a file `out.txt` which you can then parse to extract the run ID of the uploaded run.

```
polyspace-access $login -upload results/Folder/Path -parent-project myProject -output out.txt
runID=$(grep -oP '(?<=RUN_ID )\d+' out.txt)
```

Here:

- The `grep` expression extracts the digits after the string "RUN_ID " in file `out.txt`. The content of that file looks similar to this:

```
Upload with IMPORT_ID 1640263976711_d8b0fc8b-edfe-41c4-b718-6fd4b930e910.zip
Upload successful for RUN_ID 14970 and PROJECT_ID 5145
```

- `$login` is a variable that stores the login credentials and other connection information. To configure this variable, see "Encrypt Password and Store Login Options in a Variable".

If you use DOS commands, you can extract the run ID by using a `for` loop:

```
for /f "skip=1 tokens=5" %i in (out.txt) do set runID=%i
```

The loop skips the first line of the file and extracts the fifth space-delimited element (token) in the second line.

- 2 Add the Jenkins build number as a label to the project run that you uploaded. You can obtain the Jenkins build number for the Jenkins environment variable `BUILD_NUMBER`. Run this command:

```
polyspace-access $login -add-label $BUILD_NUMBER -run-id $runID
```

To add additional labels to the project run, execute the command again for each label. You cannot specify the `-add-label` option more than once each time you execute the command.

If the label that you specify for addition to a project run is already assigned to that run, the command is ignored.

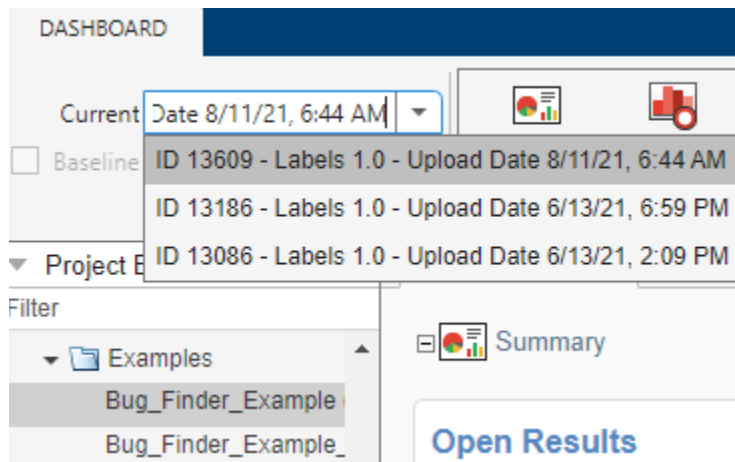
To remove a label, use the `polyspace-access` command with the `-remove-label` option. For example, to remove the label you added in step 2, use this command:

```
polyspace-access $login -remove-label $BUILD_NUMBER -run-id $runID
```

If the label that you specify for removal from a project run does not match any of the labels assigned to that run, the command is ignored.

Compare Results in Polyspace Access Project to Previous Runs and View Trends

When you open Polyspace analysis results in the Polyspace Access **DASHBOARD** or **REVIEW**, you see a snapshot of the most recent run that was uploaded to the project. To view a snapshot from an earlier run, select that run from the **Current** run drop-down list.



Select a previous run to see the state of your project from a few submissions ago. For instance, you might want to investigate a spike in findings in a previous version of your project. When you view an older project run in the **DASHBOARD** or **REVIEW** views, all the information for the currently selected run is displayed, except:

- The **Quality Objectives** settings and the **Review History** pane show the same information no matter which run you select.
- You cannot edit the **Result Details** fields if the selected run is not the latest run.

If you share a finding URL from an older run, the Polyspace Access interface opens that finding in the most recent version of the project. If the finding is not present in the most recent run, through the interface, you can open the finding in the older run.

Comparison Mode in the Polyspace Access Interface

To compare two runs in from the same project in the Polyspace Access interface, on the toolbar, select a **Current** run and a **Baseline** run. Check that the **Baseline** checkbox is enabled. You can compare current runs to only older baseline runs.

The screenshot displays the 'DASHBOARD' view of a software tool. The top navigation bar includes 'Project Overview', 'Defects', 'Code Metrics', 'Custom Rules', and 'Polyspace Guidelines'. The main content area is titled 'Project Overview' and shows a comparison between a 'Baseline Run' and a 'Current Run' for a project named 'Bug_Finder_Example'. The comparison table shows metrics such as 'Number of Files', 'Number of Lines Without Comm...', 'Defects - Total', 'Defects - Density', 'Coding Standards - Total', and 'Coding Standards - Density'. Below this, a 'Details' table provides a breakdown of findings by status: Resolved, New, and Unresolved.

Comparison		Baseline Run	Current Run
Number of Files		14	14
Number of Lines Without Comm...		5201	5201
Defects - Total		242	-
Defects - Density		36	0
Coding Standards - Total		49	-
Coding Standards - Density		9	0

Name	Resolved	New	Unre
Defects	188	-	
Custom Rules	45	-	
Polyspace Guidelines	4	-	

In the **DASHBOARD** view, the comparison shows a summary of statistics for each run and details of the number of findings that are:

- **Resolved** — Findings from the baseline run that are **Done** in the current run, or findings that are not in the current run because they are **Fixed**.
 - Findings are **Done** if they have a status of Justified, No Action Planned, or Not A Defect.
 - Findings are **Fixed** if they are fixed in the source code or the source code containing the finding is deleted or no longer part of the analysis.
- **New** — Findings that are in the current run but not in the baseline run.
- **Unresolved** — Findings that are in the baseline run and the current run.

The comparison mode is not available for the **Code Metrics** and **Quality Objectives** dashboards.

Click a cell in the **Details** table to open the corresponding results in the **Results List**.

- The total number of findings displayed in the **Results List** corresponds to the findings from the **Current** run and the findings from the **Baseline** run that are **Fixed** in the **Current** run.
- If a finding is **Resolved**, the interface displays the **Source Code** and **Result Details** information from the **Baseline** run.

In the **REVIEW** view, in addition to **Resolved**, **New**, and **Unresolved**, you can filter findings by **Fixed**. These findings are no longer in the current run because they are fixed, or the source code containing the findings is deleted or no longer part of the analysis.

To turn off the comparison mode, deselect the **Baseline** checkbox or select **None** in the **Baseline** drop-down list.

Comparison Mode at the Command Line

To compare two runs from the same project at the command-line, use the `polyspace-access -export` command and specify the run ID of a current run, the run ID of an earlier run that you use as a baseline, and the resolution type that you want to use for the comparison.

When you specify a baseline to compare with the current run, the run ID that you specify for the baseline run must exist and must point to a run in the same project as the current run.

The command generates a file with a list of findings filtered by one of these resolution types:

- **New** — Findings that are in the current run but not in the baseline run.
- **Fixed** — Findings that are fixed in the current run, either because the finding was fixed in the source code, or because the source code containing the finding is deleted or no longer part of the analysis.
- **Unresolved** — Findings from the baseline run that are still present in the current run.
- **Resolved** — Findings that are **Fixed** in the current run or findings with a status of **Justified**, **No Action Planned**, or **Not A Defect** in the current run.

For example, to compare the latest run of project `public/Bug_Finder_Example(Bug Finder)` to an earlier run:

- 1 Use the `polyspace-access -list-runs` to obtain the run IDs of the runs that you want to compare:

```
polyspace-access $login -list-runs "public/Bug_Finder_Example(Bug Finder)"

Connecting to https://example-access-server:9443
Connecting as jsmith
PROJECT_PATH "public/Bug_Finder_Example(Bug Finder)" RUN_ID 28
PROJECT_PATH "public/Bug_Finder_Example(Bug Finder)" RUN_ID 29
PROJECT_PATH "public/Bug_Finder_Example(Bug Finder)" RUN_ID 30
PROJECT_PATH "public/Bug_Finder_Example(Bug Finder)" RUN_ID 124
PROJECT_PATH "public/Bug_Finder_Example(Bug Finder)" RUN_ID 125
```

Here, `$login` is a variable that stores the login credentials and other connection information. To configure this variable, see “Encrypt Password and Store Login Options in a Variable”.

- 2 Use the command `polyspace-access -export` and specify:

- The run ID of a current run.
- The run ID of an earlier run that you use as a baseline.
- The resolution type that you want to use as a filter.

For instance, to compare the last run (run ID 125) to the second run (run ID 29) and export findings that are **Fixed**, enter this command:

```
polyspace-access $login -export 125 -baseline 29 -resolution Fixed -output ./diff_fixed.txt

Connecting to https://example-access-server:9443
Connecting as jsmith
Exporting results from RunId 125 and comparing to RunId 29
Command Completed
```

The command exports the list of findings that are fixed in the current run compared to the baseline run to file `diff_fixed.txt`.

You cannot specify more than one resolution type when you execute the command. To compare project runs for multiple resolution types, run the `polyspace-access -export` command for each resolution type.

See Also

Related Examples

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 26-2
- “Filter Results” on page 27-9

Export Results from Polyspace Access Web Server

Open or Export Results from Polyspace Access

Polyspace Access offers a centralized database where you can store Polyspace analysis results for sharing with your team and collaborative reviews. After you upload analysis results to the database, you can:

- View the results in your web browser.
- Open the results from any Polyspace desktop interface that is configured for Polyspace Access
- Export a list of results to a tab-separated value (TSV) file for further processing, such as applying custom filters and pass/fail criteria.
- Download results by using the `polyspace-access`. You use these downloaded results to merge review information between Polyspace Access projects, or as part of the baseline workflow in Polyspace as You Code. See also:
 - “Import Review Information from Existing Polyspace Access Projects” on page 26-5
 - “Baselining in Polyspace as You Code”

You cannot open results that you download with `polyspace-access` in any Polyspace interface.

The rest of this topic discusses how to open Polyspace Access results in a desktop interface and how to export results to a TSV file.

Open Polyspace Access Results in a Desktop Interface

Before you open Polyspace Access results in a desktop interface, you must configure the Polyspace desktop interface to communicate with Polyspace Access. See “Register Polyspace Desktop User Interface”.

To open results stored in the Polyspace Access database, go to **Access > Open Result** in the desktop interface, and follow the prompts. If you get a login request, use your Polyspace Access login credentials.

You can also open the desktop interface from the Polyspace Access web interface. On the toolstrip, click **Open in Desktop**. The desktop interface opens and shows the analysis results currently displayed in the Polyspace Access web interface.

Note In Linux, the desktop interface must already be open before you can view analysis results currently open in Polyspace Access.

Once you open results in the Polyspace desktop interface, changes you make to the **Status**, **Severity**, or comments fields are reflected in Polyspace Access after you save those changes.

If you open a local copy of the results that you uploaded to Polyspace Access in the desktop interface, the review fields in the **Result Details** pane are read-only. You cannot edit the **Status**, **Severity**, or comments fields.

Export Polyspace Access Results to a TSV File

You can export Polyspace Access results to a tab-separated values (TSV) file only from the command line by using the `polyspace-access` binary. When you export results, you generate a TSV file that

lists results with most of the same results attributes as the “Results List in Polyspace Access Web Interface” on page 25-19. Each listed result also includes a URL through which you can open the result in Polyspace Access. To filter the list of results you export, see the `polyspace-access export` options.

For example, to export all coding rules with status `Unreviewed` from project **myProject** stored in the **public** folder on Polyspace Access, open a command prompt terminal and enter:

```
polyspace-access -host hostName -port port ^  
-export public/myProject -coding-rules -review-status Unreviewed ^  
-output coding_rules.tsv
```

The command prompts you for your Polyspace Access login credentials, and then outputs file `coding_rules.tsv`.

hostName and *port* correspond to the host name and port number you specify in the URL of the Polyspace Access interface, for example `https://hostName:port/metrics/index.html`. If you are unsure about which host name and port number to use, contact your Polyspace Access administrator. Depending on your configuration, you might also have to specify the `-protocol` option in the command. See “Configure and Start the Cluster Admin”.

See Also

`polyspace-access`

Related Examples

- “Add Custom Status in Polyspace Access Project” on page 26-3
- “Import Review Information from Existing Polyspace Access Projects” on page 26-5

Generate Report and Variables List from Polyspace Access

Note To generate reports of results on Polyspace Access at the command line, you must have a Polyspace Bug Finder Server or Polyspace Code Prover Server installation.

Suppose that you want to generate a report and export the variables list from the results of a Code Prover analysis that you can view in a Polyspace Access project.

To connect to Polyspace Access, provide a host name and your login credentials including your encrypted password. To encrypt your password, use the `polyspace-access` command and enter your user name and password at the prompt.

```
polyspace-access -encrypt-password
login: jsmith
password:
CRYPTED_PASSWORD LAMMEACDMKEFELKMNDCONAPECEEKPL
Command Completed
```

Store the login and encrypted password in a credentials file and restrict read and write permission on this file. Open a text editor, copy these two lines in the editor, then save the file as `myCredentials.txt` for example.

```
-login jsmith
-encrypted-password LAMMEACDMKEFELKMNDCONAPECEEKPL
```

To restrict the file permissions, right-click the file and select the **Permissions** tab on Linux or the **Security** tab on Windows.

To select a project to summarize in Polyspace Access, specify the run ID of the project. To obtain a list of projects with their latest run IDs, use the command `polyspace-access` with option `-list-project`.

```
polyspace-access -host myAccessServer -credentials-file myCredentials.txt -list-project
Connecting to https://myAccessServer:9443
Connecting as jsmith
Get project list with the last Run Id
Restricted/Code_Prover_Example (Code Prover) RUN_ID 14
public/Bug_Finder_Example (Bug Finder) RUN_ID 24
public/CP/Code_Prover_Example (Polyspace Code Prover) RUN_ID 16
public/Polyspace (Code Prover) RUN_ID 28
Command Completed
```

For more information on this command, see `polyspace-access`.

Generate a Developer report for results with run ID 16 from the Polyspace Access instance with host name `myAccessServer`. The URL of this instance of Polyspace Access is `https://myAccessServer:9443`.

```
SET template_path=^
"C:\Program Files\Polyspace\R2019a\toolbox\polyspace\psrptgen\templates"

polyspace-report-generator -credentials-file myCredentials.txt ^
-template %template_path%\Developer.rpt ^
-host myAccessServer ^
-run-id 16 ^
-output-name myReport
```

The command creates report `myReport.docx` by using the template that you specify. The report is stored in folder `Polyspace-Doc` on the path from which you called the command.

Generate a tab-delimited text file that contains a list of global variables in your code for the specified analysis results.

```
polyspace-report-generator -credentials-file myCredentials.txt ^  
-generate-variable-access-file ^  
-host myAccessServer ^  
-run-id 16
```

The list of global variables `Variable_View.txt` is stored in the same folder as the generated report. For more information on the exported variables list, see “View Exported Variable List” (Polyspace Code Prover).

Review Results in IDEs

Review Results in Polyspace as You Code

Run Polyspace as You Code in Visual Studio and Review Results

You can choose to run Polyspace as You Code on each save in the Visual Studio IDE, or at will. The analysis runs on the file that is currently active in the IDE (the file must be part of a Visual Studio project, which can be part of a larger Visual Studio solution). After analysis, you see bugs and coding standard violations as source code markers or in a separate list.

Confirm Installation of Extension

To confirm that your Visual Studio installation has the Polyspace as You Code extension, check the list of extensions installed.

- In Visual Studio 2019, select **Extensions > Manage Extensions**.
- In Visual Studio 2017, select **Tools > Extensions and Updates**.

You can also confirm that the extension starts as expected on the **Output** pane. Select **View > Output** and then from the dropdown, select **Polyspace**. If the extension starts without errors, you see a message such as:

```
11/25/2020 3:59:37 PM.005: Please wait while Polyspace Connector is starting on port '9091'...
11/25/2020 3:59:41 PM.229: Polyspace Connector has started successfully.
```

The Polyspace Connector is an internal server that handles communication between the Polyspace as You Code analysis engine and the Visual Studio extension. If the default port is not available, the extension increments the port number and attempts to start the Polyspace Connector on the next port. If you use multiple Visual Studio instances, you can run Polyspace as You Code on all the instances. The Polyspace Connector in each instance uses a different port.

Run Analysis on Save

By default, Polyspace as You Code is configured to run analysis on save. Analysis results appear within a few seconds but in case of an error, you can check the progress of analysis on the **Output** pane.

After analysis, results appear as source code markers (lines below source code tokens). You also see the error locations as red circles in the scroll bar on the left.

To disable analysis on save:

- 1 Select **Tools > Options**.
- 2 On the **Polyspace** node, in the **Analysis launch mode** section, select **Manually**.

If results do not appear, see “Troubleshoot Failed Analysis or Unexpected Results in Polyspace as You Code” on page 11-81.

Run Analysis on Demand

You can also explicitly start a Polyspace analysis. To start an analysis, right-click a source file in the Visual Studio **Solution Explorer** or right-click on the source file content itself, and select **Run Polyspace analysis**.

Review Results

After analysis, the results appear in two forms:

- As source code markers (with a line below source code tokens).

You can hover on a source code token to see more details about a result.

```

file.c  x
testProject
2  int buf[MAXBUF];
3
4  int saturateAndShift(int limit, int* stream, int size) {
5      int i;
6      int numMax = 0;
7
8      if(size > MAXBUF) {
9          return -1;
10     }
11
12     if(size <= MAXBUF) {
13         for(i=0; i<size; i++) {
14             if(stream[i] > limit || stream[i] < 0) {
15                 buf[i+1] = 0;
16             }
17             else if
18                 buf
19                 numMax ++;

```

- In a list on the **Polyspace Results List** pane.

To open the pane, select **View > Other Windows > Polyspace Results List**.

Polyspace Results List						
Project: testProject File: file.c Found: 5						
Family	Ln	Ch	Check	Type	Group	Information
○	12	5	Useless if	Defect	Data flow	Impact: Medium
○	15	20	Array access out of bounds	Defect	Static memory	Impact: High
○	17	31	Invalid use of = operator	Defect	Programming	Impact: Medium
○	18	20	Array access out of bounds	Defect	Static memory	Impact: High
○	22	20	Array access out of bounds	Defect	Static memory	Impact: High

The results list shows results only for the file that is currently active in the IDE. For instance, if you switch to another file, the results list shows defects found in the new file that is active.

If you select a result in this list, you see further details of the result on the **Polyspace Result Details** pane.

Polyspace Result Details			
○ Array access out of bounds (Impact: High) ?			
Attempt to access to array element 20. Valid index range: [0 .. 19].			
Id	Event	File	Scope
1	○ Array access out of bounds	file.c	saturateAndShift()

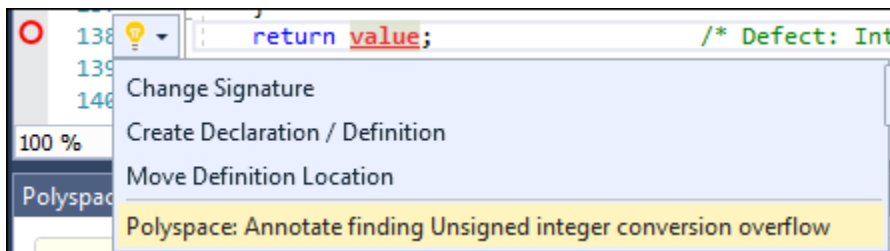
Note If you rename a file in the **Solution Explorer**, the Polyspace as You Code extension treats the renamed file as a new file and does not show previous results for this file. To view results for the renamed file, run an analysis.

Justify Results Using Code Annotations

If you decide not to fix a result, you can add code annotations to the result to avoid another review. If the annotations follow a specific syntax, subsequent Polyspace as You Code runs can read these annotations and suppress the corresponding results.

To add a code annotation, click the source code token containing a result. Click the light bulb icon that appears and select **Polyspace Annotate finding *result_name***. The annotation is entered on the same line as the result.

If you import review information from a baseline run, the code annotation includes any status, severity, or comment assigned to that result in the baseline run. If the imported status is set to **To investigate**, **To fix**, or **Other**, the annotation does not suppress the result.



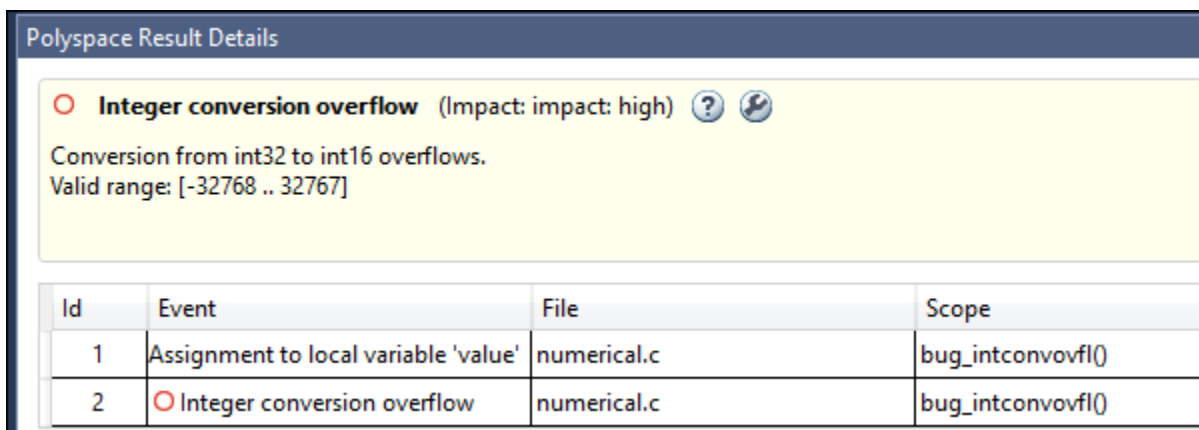
See also:

- “Annotate Code and Hide Known or Acceptable Results” on page 30-2
- “Short Names of Bug Finder Defect Groups and Defect Checkers” on page 30-11

View Help

You can see more information on a type of result by visiting the context-sensitive help page for the result.

- To open the context-sensitive help for a result, first open the **Polyspace Result Details** pane for a result. Then, click the question mark icon next to the result details.
- To navigate directly to the **Fix** section of the context-sensitive help for a result, click the wrench icon next to the result details.



You can also open the full searchable documentation for the Polyspace as You Code extension in your system browser. To open the documentation, select **Help > Open Polyspace Product Help**.

Configure Checkers and Other Settings

By default, Polyspace as You Code checks for defects that are likely to be of most interest to developers. You can expand the set of checkers and perform other configuration through the Polyspace as You Code extension settings in Visual Studio. To open the settings, select **Tools > Options** and specify appropriate settings on the **Polyspace** node.

For instance, you might want to:

- Enable or disable certain checkers.

See “Configure Checkers for Polyspace as You Code in Visual Studio” on page 11-63.

- See only new results.

See “Baseline Polyspace as You Code Results in Visual Studio” on page 11-44.


For the full list of settings, see “Configure Polyspace as You Code Extension in Visual Studio” on page 11-2.

Run Polyspace as You Code in Visual Studio Code and Review Results

You can choose to run Polyspace as You Code on each save in the Visual Studio Code editor, or at will. The analysis runs on the file that is currently active in the editor. After you run the analysis, you see bugs and coding standard rule violations as source code markers and as a list in the **Problems** panel.

Check Installation and Start Extension

To confirm that the Polyspace as You Code extension is installed in Visual Studio Code, check the **INSTALLED** section in the **Extensions** view.

To open this view, in Visual Studio Code, select **View > Extensions** or click  in the activity bar on the left and look for Polyspace as You Code in the list of installed extensions.

To start the Polyspace as You Code extension, open a C or C++ file in the editor. The extension does not start automatically when you open Visual Studio Code when no C or C++ file is open in the editor. To check the status of the extension, do one of the following:

- Open the **Problems** panel and select **Polyspace as You Code** from the dropdown on the right.
- Place your cursor over the icons in the status bar on the lower left. See “View Extension Information in Status Bar” on page 29-6.
- Open the Polyspace views. See “Open Additional Polyspace Views” on page 29-7.

Look for Polyspace as You Code in the list of extensions installed.

Note

- If you use Visual Studio Code version 1.57.1 or later, the Polyspace as You Code extension is disabled when you open a folder in **Restricted Mode**. To enable the extension, mark the folder as trusted. See Workspace Trust.
 - For better performance on Windows systems, the files that you analyze should not be stored on network drives.
-

View Extension Information in Status Bar


Once you start the extension, Polyspace as You Code displays icons in the status bar (lower left) to provide information about the state of the extension, such as errors in the configuration or whether an analysis is running on the currently selected file.

- Place your cursor over an icon to view a tooltip containing additional information.
- Click an icon (when applicable) to perform common operations, such as opening the extension settings or viewing the **OUTPUT** panel.

The information that you see in the status bar is either global (🌐) or specific to the currently selected file (📄). The file specific status bar icons update when you select a different file.

To disable or reenable the Polyspace as You Code status bar icons, right-click the icons and select the appropriate option from the menu.

Open Additional Polyspace Views

The Polyspace as You Code extension provides different views in the side bar that show information about the state of your configuration and of the files that you select for analysis. Click  in the activity bar on the left to show or hide these Polyspace views.

QUALITY MONITORING View

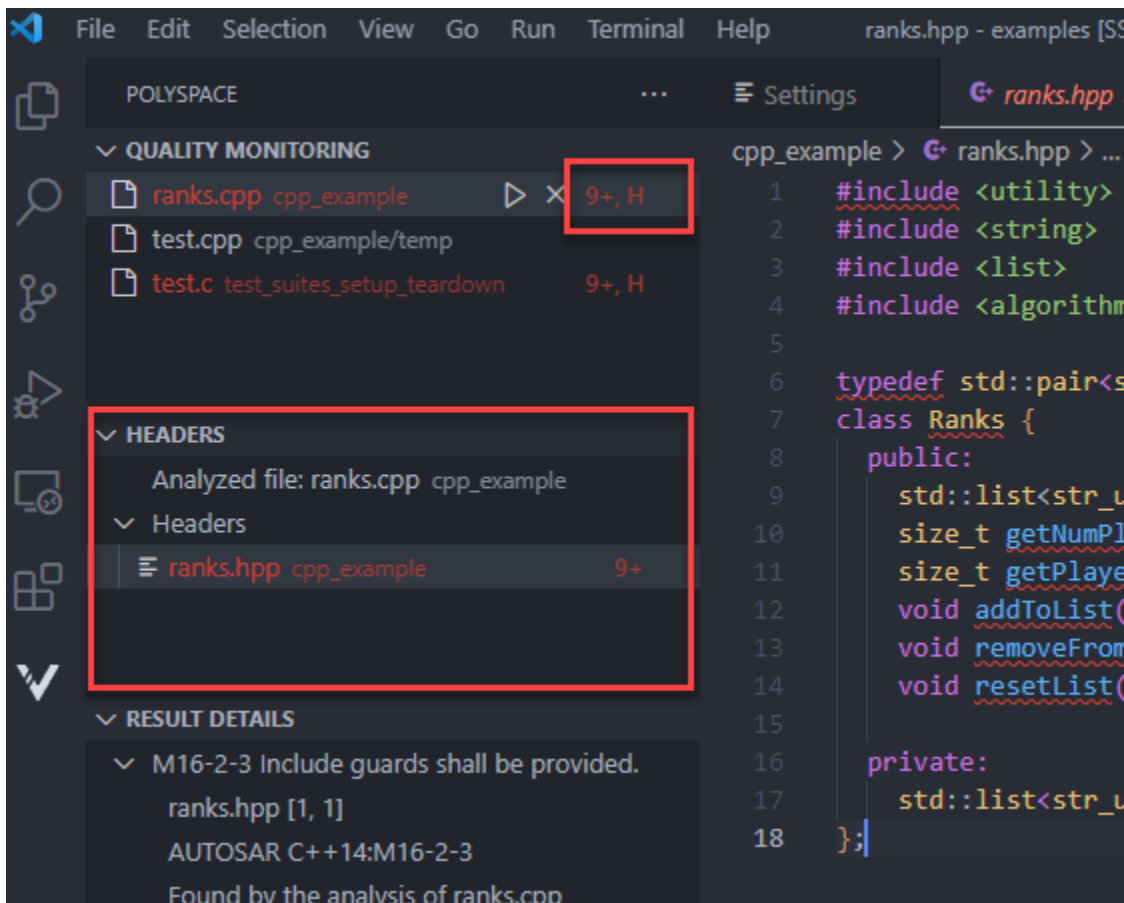
This view shows all the files that you have selected for analysis, the status of the analysis for each file, and the number of findings.

In this view, you can also start a file-by-file analysis of all the files that you added to the view or you can choose to analyze only a specific file.

Use the **Analysis Options: Add To Quality Monitoring On Save** setting to enable or disable adding files to this view on save (Ctrl + S). See “Analysis Behavior On Save” on page 11-9. To add a file manually, right-click the file in the editor, the **EXPLORER** view, or **SOURCE CONTROL** view in the side bar.

HEADERS view

When you analyze a file, Polyspace also analyzes header files that are included by the analyzed file, if those headers are in the same folder as the analyzed file. If Polyspace reports findings in a header file, you see an **H** next to the analyzed file in the **Quality Monitoring** view. Click the analyzed file in that view to list the header in the **HEADERS** view.





Header files are listed only if they contain findings. Click a header file in the **HEADERS** view to open its finding in the **PROBLEMS** panel and start inspecting those findings in the **Editor**.

RESULT DETAILS View

When you select a finding in the **OUTPUT** panel or in the editor, this view shows additional information about the finding, including, when available, the traceback and the review information imported from the baseline.

If you select multiple lines of code, the view lists additional details for all the findings reported in the selected code.

Click  to open the contextual help for a finding.




Whenever you select a finding, the **RESULT DETAILS** view updates to show information about the selected finding. To lock the view and keep it from updating when you browse through findings in the source code or the **PROBLEMS** panel, click the  icon in the upper right. To unlock the view, click .

BASELINE View

View the current mode for the baseline (whether you are using the baseline or not) and, if you use the baseline, the status of the baseline download. If you have configured the baseline settings, you see

information about the Polyspace Access server and the project you use as a baseline. See “Baseline Polyspace as You Code Results in Visual Studio Code” on page 11-48.



In this view, you can:

- Click  to download a baseline.
- Click  to open the baseline project in the Polyspace Access interface
- Click  to open the baseline settings.

If the results from the baseline run are generated with a Polyspace product that is older than your version of Polyspace as You Code, this view shows a warning.

CONFIGURATION View

This view shows your current **Analysis Options** settings and the status of the generated build options file, if applicable.

- Click  to open the **Analysis Options** settings. See “Analysis Setup” on page 11-9.
- Click  to open the **Checkers selection** interface. See “Configure Checkers for Polyspace as You Code in Visual Studio Code” on page 11-66.

Run Analysis

On Save

By default, Polyspace as You Code adds the current file to the **Quality Monitoring** list and analyzes this file on save. Analysis results appear within a few seconds on the source code. In case of an error, you see a popup with the error message. To diagnose further, select **View > Output**. On the **OUTPUT** panel, from the dropdown on the upper right, select **Polyspace as You Code**.

After analysis, results appear as source code markers (wavy lines below source code tokens). You also see the error locations as red marks on the scroll bar. Click an error location to navigate to the corresponding source code.

To disable analysis on save:

- 1 Open the settings by pressing **Ctrl + ,** (comma) and type `polyspace.analysisOptions.OnSave` in the settings search bar.
- 2 Deselect the setting for **Analysis Of Files On Save**.

If results do not appear, see “Troubleshoot Failed Analysis or Unexpected Results in Polyspace as You Code” on page 11-81.

On Demand

You can also explicitly start a Polyspace analysis. To start an analysis, do one of the following:

- Right click a source file in the **EXPLORER** view or right-click on the file content itself in the editor and select **Run Polyspace Analysis**.
- Click the play button in the **QUALITY MONITORING** view.

- With your cursor in the source file, press **Ctrl + Shift + Alt + A**.

Review Results

After analysis, the results appear in two forms:


- As source code markers (with a wavy line below source code tokens).

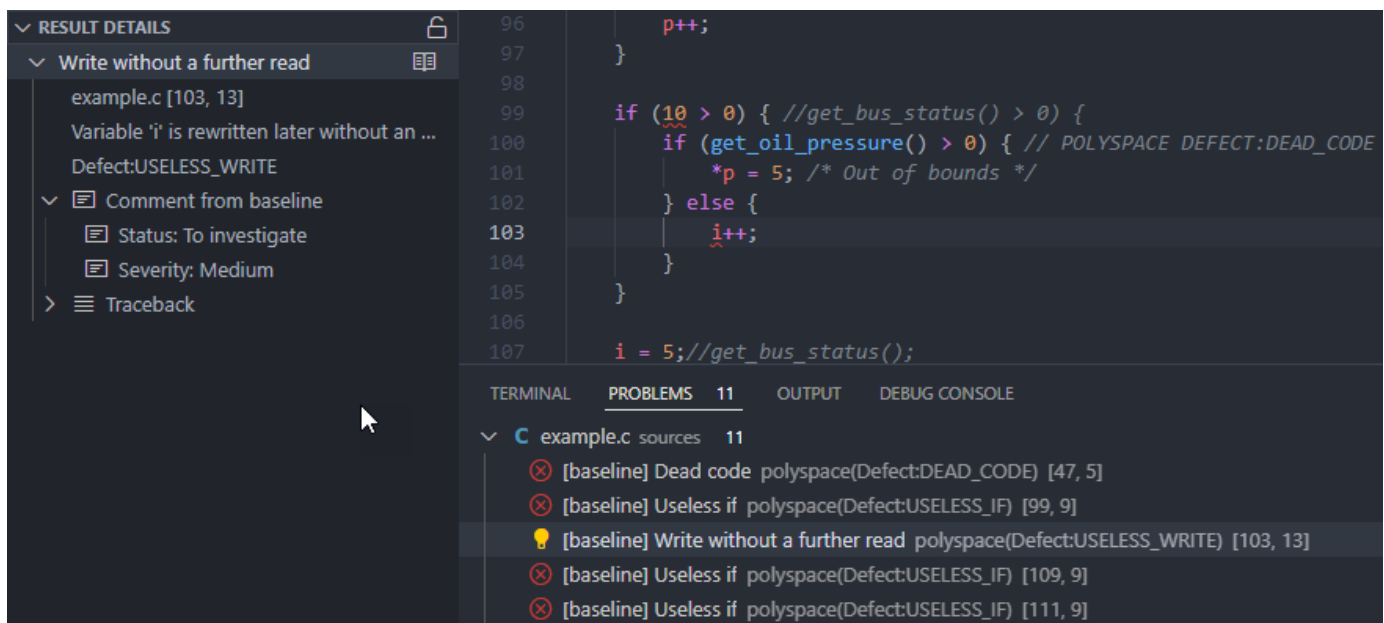
You can hover on a source code token to see more details about a result.

- In a list on the **PROBLEMS** panel.

To open the panel, select **View > Problems**.

When you select a finding you can view additional details about the result or an event trace in the **RESULT DETAILS** view.

Click  in this view to open the contextual help for this finding.



Apply Suggested Fix for Common Defects or Coding Rule Violations

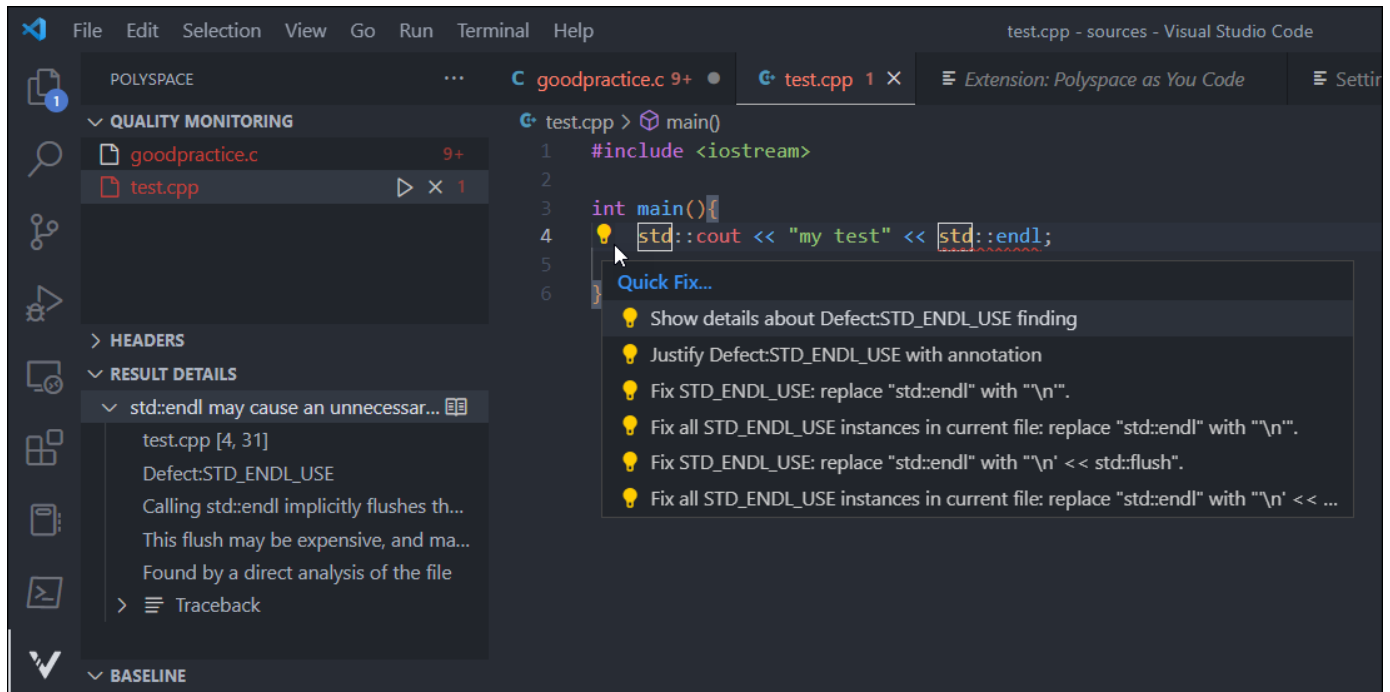
If Polyspace reports certain common defects or coding rule violations in your code, the Polyspace as You Code extension proposes one or more fixes you can apply to these findings.

This table lists the common defects and coding rule violations for which Polyspace proposes a fix, along with the Polyspace checkers that report these errors.

Defect or Coding Rule Violation	Polyspace Checkers
Function that could be evaluated a compile time missing constexpr specifier.	Missing constexpr specifier

Defect or Coding Rule Violation	Polyspace Checkers
Unmodified expensive-to-copy parameter passed by value.	AUTOSAR C++14 Rule A8-4-7 Expensive pass by value
Unmodified variable not declared const.	AUTOSAR C++14 Rule A7-1-1 CERT C: Rec. DCL00-C MISRA C++:2008 Rule 7-1-1 Unmodified variable not const-qualified
Unused include	AUTOSAR C++14 Rule A16-2-2 Useless include
Unused parameter	AUTOSAR C++14 Rule A0-1-4 CERT C: Rec. MSC13-C MISRA C:2012 Rule 2.7 Unused parameter
Use of <code>std::endl</code>	<code>std::endl</code> may cause an unnecessary flush

To apply a quick fix, select a finding in the editor or the **Problems** panel, and then, from the light bulb menu, click the appropriate option. The quick fix options have the format **Fix *checkerName*: *proposedFix***. Here, *checkerName* is the name of the selected defect checker or coding rule, and *proposedFix* is the fix that Polyspace applies to your code. Polyspace edits your code and fixes the issue or all instances of the issue in the currently active file, depending on the option that you select. If there is more than one proposed fix, you see all available fixes in the menu.



Justify Results Using Code Annotations

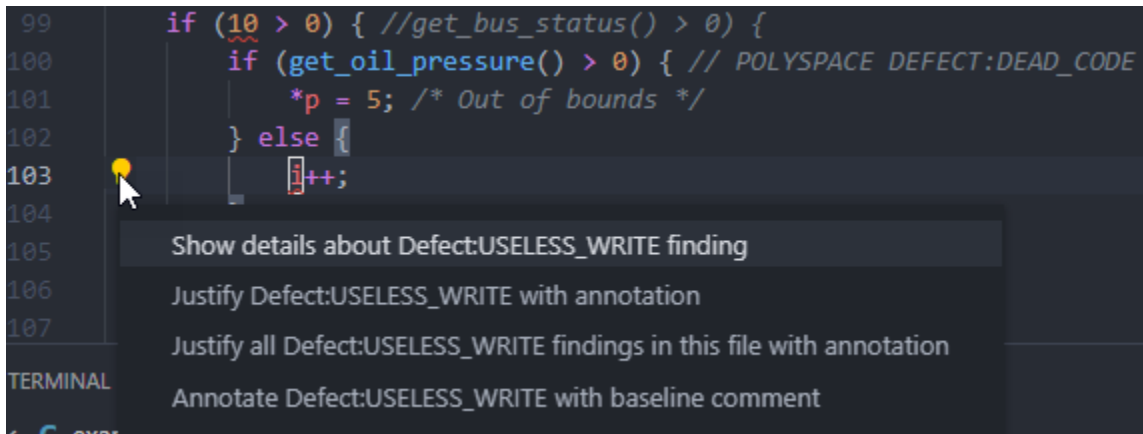
If you decide not to fix a result, you can add code annotations to the result to avoid having to fix the result again. If the annotations follow a specific syntax, subsequent Polyspace as You Code runs can read these annotations and suppress the corresponding results.

- To add a code annotation, click the light bulb icon beside the source code token containing a result and select **Justify *result_name* with annotation**. The annotation is entered on the same line as the result.
- To mass-justify findings of the same defect or coding rule violations in the current file, select an instance of that finding and click the **Justify all *result_name*** option from the light bulb menu. Polyspace inserts an annotation in your code on each line that contains this finding
- If you import review information from a baseline run, select **Annotate *result_name* with baseline comment** to include any status, severity, or comment assigned to that result in the baseline run. If the imported status is set to `To investigate`, `To fix`, or `Other`, the annotation does not suppress the result.

If you select multiple lines of code, the light bulb menu show annotation options for each finding in your selection.

Polyspace as You Code inserts code annotations as comments enclosed in `/* */` for C files and prefixed with `///
//` in C++ files (C++ style). The comment format in C files complies with MISRA C:2004 rule 2.2.

Justified results are removed from the **PROBLEMS** panel and are suppressed in subsequent analyses.




See also:

- “Annotate Code and Hide Known or Acceptable Results” on page 30-2
- “Use a Justification Catalog to Autocomplete Annotations in Polyspace as You Code plugins” on page 29-27
- “Short Names of Bug Finder Defect Groups and Defect Checkers” on page 30-11

View Context-Sensitive Help for Result

You can see more information on a type of result by visiting the context-sensitive help page for the result.

- To open the context-sensitive help for the currently selected finding, click  in the **RESULT DETAILS** view. see “Open Additional Polyspace Views” on page 29-7.
- You can also view the context-sensitive help for any Polyspace as You Code result by going to the **Command Palette (Ctrl + Shift + P)** and selecting **Polyspace: Open Contextual Help**. Choose a result family from the menu and then enter a result name, for instance `UNMODIFIED_VAR_NOT_CONST`.

To open the full searchable documentation for the Polyspace as You Code extension in your system browser, go to the **Command Palette (Ctrl + Shift + P)** and select **Polyspace: Open Documentation**.

Configure Checkers and Other Settings

By default, Polyspace as You Code checks for defects that are likely to be of most interest to developers. You can expand the set of checkers and perform other configuration through the Polyspace as You Code extension settings in Visual Studio Code.

Open the settings by pressing **Ctrl + ,** (comma) and type `polyspace` in the settings search bar.

For instance, you might want to:

- Enable or disable certain checkers.

See “Configure Checkers for Polyspace as You Code in Visual Studio Code” on page 11-66.

- See only new results.

See “Baseline Polyspace as You Code Results in Visual Studio Code” on page 11-48.

For the full list of settings, see “Configure Polyspace as You Code Extension in Visual Studio Code” on page 11-8.

Run Polyspace as You Code in Eclipse and Review Results

This topic describes how to run a single-file analysis in Eclipse using Polyspace as You Code. For Polyspace desktop products such as Polyspace Bug Finder, see “Bug Finder Analysis Based on Eclipse Projects”

You can configure Polyspace as You Code to run on each save in your Eclipse-based IDE, or you can start the analysis manually. The analysis runs on the file that is currently active in the IDE. After analysis, you see bugs and coding standard violations as source code markers or in a separate list.

Check Installation and Start Plugin

To confirm that the Polyspace as You Code plugin is installed in your Eclipse-based IDE, check the list of installed plugins.

- 1 Select **Help > About Eclipse IDE**.
- 2 Click **Installation Details** and type polyspace in the search bar on the **Plug-ins** tab.

To start the Polyspace as You Code plugin, open a C or C++ file in the editor. The plugin does not start automatically when you open the Eclipse IDE when no C or C+ file is open in the editor. You can confirm that the extension starts as expected in the **Console** view. To open this view, select **Window > Show View > Console**. If the plugin starts without errors, you see a message similar to this:



```
[2023-01-26 00:43:42.68] [Information] No setting set for the port. Trying to find a free port.  
[2023-01-26 00:43:49.539] [Information] Connector was successfully started on port 56041
```


You can also open the **Console** view by pressing **Alt+Shift+Q** and then **Q** again, and start typing console in the search bar.

The Polyspace as You Code plugin establishes an internal connection with the analysis engine on the selected port. If you do not specify a port, Polyspace queries your system for an open port and uses the port that your system returns.

Open Polyspace as You Code Perspective



The Polyspace as You Code plugin has a dedicated perspective in the Eclipse IDE that groups all the relevant Polyspace views.


To open the Polyspace as You Code perspective, click  on the far right of the tool bar. If the perspective is already open but a different perspective is active, click  to return to the Polyspace perspective.

The different views in the Polyspace as You Code perspective show information about the state of your configuration and about the files that you select for analysis. For more information, click  in each view (when available) to open the documentation.

Quality Monitoring View

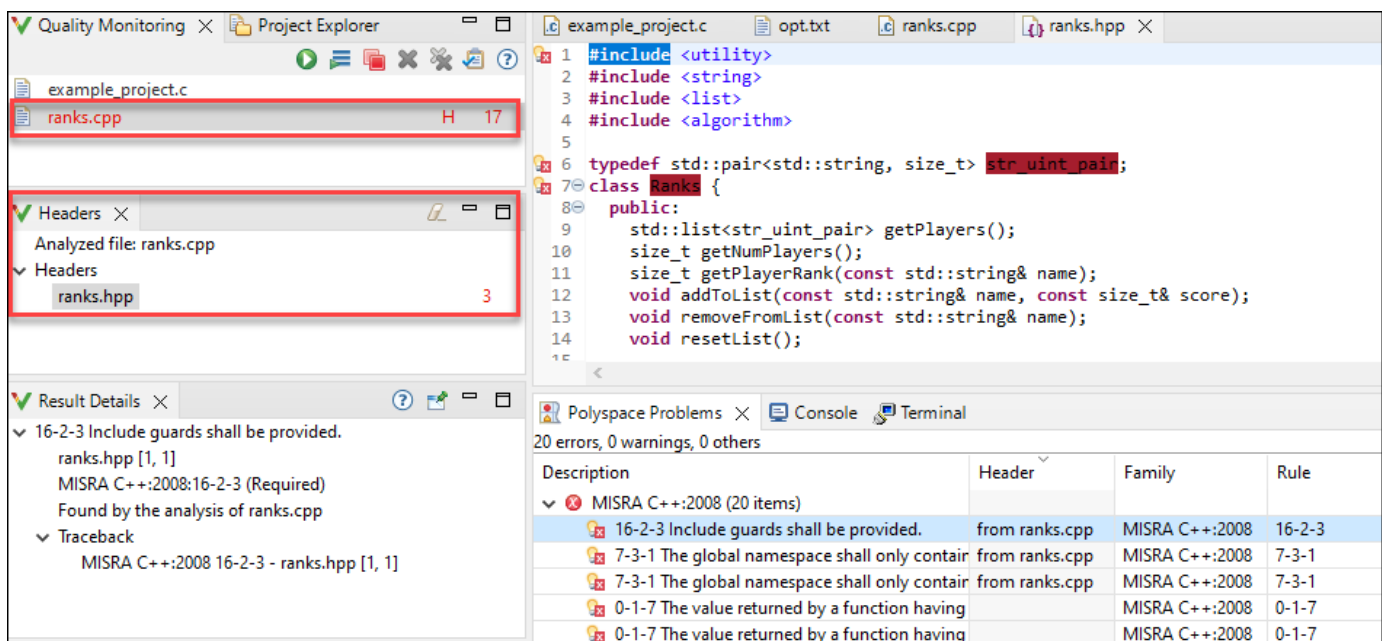
This view shows all the files that you have selected for analysis, along with the status of the analysis and the number of findings for each file.

To analyze a file, select that file and click . You can also start a sequential file-by-file analysis by clicking .

Open the **Analysis** node setting by clicking  to configure adding files to this view save. See “Plugin Behavior on Save” on page 11-20. To add a file manually, right-click the file in the editor or in the **Project Explorer**.

Headers View

When you analyze a file, Polyspace also analyzes the header files that are included by and that are in the same folder as the analyzed file. If Polyspace reports findings in a header file, you see an **H** next to the analyzed file in the **Quality Monitoring** view. Click the analyzed file in that view to list the headers file in the **Headers** view.



The screenshot shows the Polyspace IDE interface. The **Project Explorer** on the left shows the file `ranks.cpp` with a red box around it and an **H** icon next to it. Below it, the **Headers** view is open, showing the analyzed file `ranks.cpp` and a list of headers, with `ranks.hpp` selected and a red box around it. The **Result Details** view shows the analysis results for `ranks.cpp`, including the MISRA C++:2008 16-2-3 rule. The **Polyspace Problems** view at the bottom shows a table of 20 errors, 0 warnings, and 0 others.

Description	Header	Family	Rule
20 errors, 0 warnings, 0 others			
MISRA C++:2008 (20 items)			
16-2-3 Include guards shall be provided.	from ranks.cpp	MISRA C++:2008	16-2-3
7-3-1 The global namespace shall only contain	from ranks.cpp	MISRA C++:2008	7-3-1
7-3-1 The global namespace shall only contain	from ranks.cpp	MISRA C++:2008	7-3-1
0-1-7 The value returned by a function having		MISRA C++:2008	0-1-7
0-1-7 The value returned by a function having		MISRA C++:2008	0-1-7

Header files are listed only if they contain findings. Click the file in the **Headers** view to open its findings in the **Polyspace Problems** view.

Polyspace Problems View


This view shows all the analysis findings listed as one finding per line. Each line shows additional information such as the rule description, rule name, file containing the finding, and review information. The findings are grouped by results family. To change the grouping or view a flat list, click the three vertical dots and select **Grouping options**.


Baseline	Description	Header	Family	Rule	Category	File	Line
✖	MISRA C:2012 (3 items)						
✖	17.7 The value returned by a function having non-...		MISRA C:2012	17.7	Required	example_projec...	15
✖	21.6 The Standard Library input/output functions s...		MISRA C:2012	21.6	Required	example_projec...	15
✖	Dir 4.6 typedefs that indicate size and signedness s...		MISRA C:2012	D4.6	Advisory	example_projec...	14
✖	MISRA C++:2008 (20 items)						
✖	16-2-3 Include guards shall be provided.	from ranks.cpp	MISRA C++:2008	16-2-3	Required	ranks.hpp	1
✖	7-3-1 The global namespace shall only contain ma...	from ranks.cpp	MISRA C++:2008	7-3-1	Required	ranks.hpp	6
✖	7-3-1 The global namespace shall only contain ma...	from ranks.cpp	MISRA C++:2008	7-3-1	Required	ranks.hpp	7
✖	0-1-7 The value returned by a function having a no...		MISRA C++:2008	0-1-7	Required	ranks.cpp	29
✖	0-1-7 The value returned by a function having a no...		MISRA C++:2008	0-1-7	Required	ranks.cpp	40
✖	5-0-13 The condition of an if-statement and the co...		MISRA C++:2008	5-0-13	Required	ranks.cpp	22
✖	5-0-3 A cvalue expression shall not be implicitly co...		MISRA C++:2008	5-0-3	Required	ranks.cpp	16
✖	5-0-4 An implicit integral conversion shall not cha...		MISRA C++:2008	5-0-4	Required	ranks.cpp	16
✖	5-0-4 An implicit integral conversion shall not cha...		MISRA C++:2008	5-0-4	Required	ranks.cpp	17
✖	5-19-1 Evaluation of constant unsigned integer ex...		MISRA C++:2008	5-19-1	Advisory	ranks.cpp	17

Click a column header to sort findings by the content of that column. For instance, you can sort findings by files. Findings from a header file list the source file that includes the header in the **Header** column.

Result Details View




When you select a finding in the **Polyspace Problems** view or in the editor, this view shows additional information about the finding, including (when available) the traceback and the review information imported from the baseline.

If you select multiple lines of code, the view shows the results details for all the findings in the selection. Click  to view contextual help about the selected finding.

To lock this view while you browse through findings in the source code or the **Polyspace Problems** view, click . The view does not update when you select a different finding. Click the icon again to unlock the view.

Configuration View



This view shows the current **Analysis** node settings and the status of the generated build options, if applicable.

- Click  to generate a build options file. This button is available only if you configure the plugin to extract build options from your build command, from your Eclipse C/C++ project, or from a JSON compilation database. See “Generate Build Options for Polyspace as You Code Analysis in Eclipse” on page 11-37.
- Click  to open the **Analysis** node settings. See “Analysis Node” on page 11-20.
- Click  to open the **Checkers selection** interface. See “Configure Checkers for Polyspace as You Code in Eclipse” on page 11-60.

Baseline View

View whether you are using the baseline or not, and the status of the baseline download if applicable. If you configure the baseline settings, you see information about the Polyspace Access server and the project you use as a baseline. See “Baseline Polyspace as You Code Results in Eclipse” on page 11-53.

In this view, you can:


- Click  to download or update a baseline.
- Click  to open the baseline settings.

If the results from the baseline run are generated with a Polyspace product version that is older than the Polyspace as You Code version, the view shows a warning.

Run Analysis

On Save

By default, the Polyspace as You Code plugin adds the current file to the **Quality Monitoring** view on save but does not start the analysis.


To enable the analysis of the current file on save, click  in the **Quality Monitoring** view and select **Start analysis on save** in the **Preferences** window. You can also open the preferences by pressing **Alt+W+P**.

After analysis, findings appear as source code markers. You also see the finding locations as red marks on the scroll bar. Click a finding location to navigate to the corresponding source code. If the analysis fails, check the messages in the **Console** view to diagnose the cause of the failure.

If results do not appear, see “Troubleshoot Failed Analysis or Unexpected Results in Polyspace as You Code” on page 11-81.

On Demand

To disable starting an analysis on save, go to the Polyspace as You Code preferences and unselect **Start analysis on save**. To start an analysis manually, do one of the following:

- Click  in the **Quality Monitoring** view.
- Right-click the file in the editor and click **Run Polyspace Analysis**.

Review Results

After analysis, the results appear in two forms:

- As source code markers.

You can hover on a source code marker to view more details about the finding and view options to justify that finding.

- In a list on the **Polyspace Problems** view.

If you select a finding in the **Polyspace Problems** view or in the editor, you see additional details about the finding in the **Results Details** view. Click  to open the contextual help for the currently selected finding.

Justify Results Using Code Annotations

If you decide not to fix a result, you can add code annotations to the result to avoid having to fix the result again. If the annotations follow a specific syntax, subsequent Polyspace as You Code runs can read these annotations and suppress the corresponding results.

- To add a code annotation, right-click a finding in the **Polyspace Problems** and select **Polyspace > Justify the current result**. The annotation is entered on the same line as the finding. You can also apply an annotation by hovering on the code marker for that result.
- To mass-justify findings of the same defect or coding rule violation in the current file, right-click a finding in the **Polyspace Problems** and select **Polyspace > Justify all instances of the selected finding**. Polyspace inserts an annotation in your code on each line that contains this finding.
- if you import review information from a baseline run, any status, severity, or comment from the baseline run is automatically applied to the corresponding Polyspace as You Code finding.

if the imported status is **Not a defect**, **Justified**, or **No action planned** in the baseline run are suppressed from the **Polyspace Problems** view and in subsequent analyses.

If you select multiple lines of code and then go to the right-click menu in the **Polyspace Problems** view, the **Polyspace > Justify the current result** option justifies each result in the selected code.


Polyspace as You Code inserts code annotations as comments enclosed in `/* */` for C files and prefixed with `///
//` in C++ files (C++ style). The comment format in C files complies with MISRA C:2004 rule 2.2.


See also:

- “Annotate Code and Hide Known or Acceptable Results” on page 30-2
 - “Short Names of Bug Finder Defect Groups and Defect Checkers” on page 30-11
- “Use a Justification Catalog to Autocomplete Annotations in Polyspace as You Code plugins” on page 29-27

View Context-Sensitive Help for Result

You can see more information on a type of result by visiting the context-sensitive help page for the result.

To open the context-sensitive help for a finding, select that finding in your code or in the **Polyspace Problems** view, and then click  in the **Result Details** view.

You can also open the full searchable documentation for the Polyspace as You Code plugin in your system browser by clicking  in the **Baseline**, **Configuration**, or **Quality Monitoring** view.

Configure Checkers and Other Settings

By default, Polyspace as You Code checks for defects that are likely to be of most interest to developers. You can expand the set of checkers and perform other configuration through the Polyspace as You Code plugin settings in Eclipse. To open the settings, press **Alt+W+P** and go to the **Polyspace as You Code** node in the **Configuration** window.

For instance, you might want to:

- Enable or disable certain checkers.

See “Configure Checkers for Polyspace as You Code in Eclipse” on page 11-60.

- See only new results.

See “Baseline Polyspace as You Code Results in Eclipse” on page 11-53.

For the full list of settings, see “Configure Polyspace as You Code Plugin in Eclipse” on page 11-19.

Run Polyspace as You Code from Command Line and Export Results

You can run Polyspace as You Code on source files directly at the command line.

For IDEs that are not directly supported with a Polyspace as You Code plugin, you can open a terminal within the IDE and run the commands, or create a menu item to run the commands on the file currently open in the IDE. You can even incorporate these commands in a makefile, so that building your code also runs static analysis on the code. See also “Integrate Polyspace as You Code in IDEs and Editors Without Plugins” on page 29-24.

Add Install Folder to Path

To avoid typing the full path to Polyspace as You Code commands, add the paths to these commands to the PATH environment variable on your operating system.

The paths in the default installation folder are the following:

Windows	C:\Program Files\Polyspace as You Code\R2023a\polyspace\bin
Linux	/usr/local/PolyspaceAsYouCode/R2023a/polyspace/bin

After you add the paths, you can enter commands such as the following in a terminal without errors:

```
polyspace-bug-finder-access -help
```

Run Analysis and See Results on Console

To run Polyspace as You Code, use the `polyspace-bug-finder-access` command. Export the results to the console using the `polyspace-results-export` command.

```
polyspace-bug-finder-access -sources filename
polyspace-results-export -format console
```

In this example, the `polyspace-bug-finder-access` command generates results in the current folder. The `polyspace-results-export` command reads results from the current folder and exports to the console.

The analysis typically takes a few seconds to complete (slightly longer for C++ files). If the analysis fails to complete, further details of the error appear on the console. You can use the option `-no-quiet` to see a more detailed analysis log on the console.

Store Results in Specific Folder

To use a specific results folder *resultsFolder* instead of the current folder, change the preceding lines as follows:

```
polyspace-bug-finder-access -sources filename -results-dir resultsFolder
polyspace-results-export -format console -results-dir resultsFolder
```

Export Results to JSON Format (SARIF Output)

Instead of displaying analysis results on the console, you can export the results to a JSON file. You can then parse this file using a JSON parser method in any language that you want.

```
polyspace-bug-finder-access -sources filename.c  
polyspace-results-export -format json-sarif -output-name outputFilePath
```

Here, *outputFilePath* is the full path to the JSON file.

The JSON format follows the standard notation provided by the OASIS Static Analysis Results Interchange Format (SARIF).

Specify Analysis Options by Using Options Files

To adapt the Polyspace analysis configuration to your development environment and requirements, you have to modify the default configuration through command-line options such as `-compiler`. Options files are a convenient way to collect multiple options together and reuse them across projects.

Options files are text files with one option per line. For instance, the content of an options file can look like this:

```
# Options for Polyspace analysis  
# Options apply to all projects in Controller module  
-compiler visual16.x  
-D _WIN32  
-checkers-activation-file "Z:\utils\checkers.xml"
```

Specify an options file using the option `-options-file`. For instance:

```
polyspace-bug-finder-access -sources file.c -options-file "Z:\utils\polyspace\options.txt"
```

See also “Options Files for Polyspace Analysis” on page 12-5. For all options available with Polyspace as You Code, see “Complete List of Polyspace Bug Finder Analysis Engine Options”.

Create Options File by Analyzing Build

Instead of entering options by hand in an options file, you can create an options file with all Polyspace options required for compilation by analyzing your build system. For instance, you can trace your build command and save the options in a file `buildOptions.txt` that you can use for the subsequent analysis.

```
polyspace-configure -no-sources -output-options-file buildOptions.txt buildCommand  
polyspace-bug-finder-access -sources file.c -options-file buildOptions.txt
```

Here, *buildCommand* is a build command that performs a full build of your source code, for instance, `make -B` or `make --always-make`. For build systems that can output compilation options in the JSON compilation database format, you can obtain the options from the JSON file:

```
polyspace-configure -no-sources -output-options-file buildOptions.txt -compilation-database jsonFile
```

Here, *jsonFile* is the full path to the compilation database JSON file.

You can also append a second options file with options related to the analysis such as checkers. For instance, if the second options file is called `checkersOptions.txt`, you can run Polyspace as You Code as follows:

#DOS Command

```
polyspace-bug-finder-access -sources file.c ^  
-options-file buildOptions.txt^  
-options-file checkersOptions.txt
```

#Linux Command

```
polyspace-bug-finder-access -sources file.c \  
-options-file buildOptions.txt\  
-options-file checkersOptions.txt
```

See Also

`polyspace-bug-finder-access` | `polyspace-results-export` | `polyspace-configure`

More About

- “Options Files for Polyspace Analysis” on page 12-5
- “Integrate Polyspace as You Code in IDEs and Editors Without Plugins” on page 29-24

Integrate Polyspace as You Code in IDEs and Editors Without Plugins

Polyspace as You Code supports these IDEs with extensions or plugins: Visual Studio, Visual Studio Code, and Eclipse. Even if an IDE is not explicitly supported with a Polyspace as You Code plugin, you can open a console within the IDE and run Polyspace as You Code commands, or create a menu item to run the commands on the file currently open in the IDE.

This topic demonstrates how to integrate Polyspace as You Code in a simple editor such as Notepad++. You can use the principles here to integrate Polyspace as You Code in most editors or IDEs.

Overview of Approach

In supported IDEs, a Polyspace as You Code extension allows you to analyze the file that is currently active in the IDE and see results within the IDE (as source code markers or in a list). In an unsupported IDE or editor, you can partly emulate this workflow, that is, run analysis within the IDE and view results. The workflow consists of two steps:

- *Running analysis and exporting results*

Most IDEs or editors provides environment variables that resolve to the current file path. You can create menu items that execute a script which runs the `polyspace-bug-finder-access` command on this path. In the same script, you can export the results to the IDE console.

- *Parsing console output to allow navigation to line*

Each Polyspace as You Code result in the console output starts with a line in this format:

```
filepath:lineNumber:columnNumber
```

Here, *filepath* is the path to the current file, *lineNumber* is the line number of the result, and *columnNumber* is the column that starts the token with the result. For instance:

```
C:\MyProj\myFile.c:17:31:
```

indicates that the file `C:\MyProj\myFile.c` contains a result on line 17, starting from column 31. If you can parse the console output, you can enable a navigation to line 31 to the start of the token containing the result.

Integration Steps

This example shows an integration of Polyspace as You Code in a simple editor such as Notepad++. You can follow similar integration steps in other editors such as GNU Emacs, Sublime Text, and so on.

Step 1: Set Up Script Runs from Within Editor

In Notepad++, you can use a plugin such as NppExec that allows you to execute any script from within the editor. The editor also provides the environment variable `$(FULL_CURRENT_PATH)` that resolves to the file that is currently active in the IDE.

The simplest script that can be run within the plugin can be the following:

```
cd $(CURRENT_DIRECTORY)
set POLYSPACE_EXECUTABLES_FOLDER=C:\Program Files\Polyspace as You Code\R2023a\polyspace\bin
```

```
set POLYSPACE_ENGINE=$(POLYSPACE_EXECUTABLES_FOLDER)\polyspace-bug-finder-access.exe
set POLYSPACE_REPORT_EXPORTER=$(POLYSPACE_EXECUTABLES_FOLDER)\polyspace-results-export.exe
$(POLYSPACE_ENGINE) -sources $(FULL_CURRENT_PATH)
$(POLYSPACE_REPORT_EXPORTER) -results-dir . -format console
```

In practice, you might want to specify additional analysis options using an options file. If the options file is called `polyspace_options.txt`, the command to run Polyspace as You Code in the preceding script can be replaced with:

```
$(POLYSPACE_ENGINE) -sources $(FULL_CURRENT_PATH) -options-file polyspace_options.txt
```

See also “Options Files for Polyspace Analysis” on page 12-5.

For other command-line examples, see “Run Polyspace as You Code from Command Line and Export Results” on page 29-21. For instance, instead of exporting to the console directly, you can export the results to a JSON format, use a JSON parser to package the results, and then export them to the console or use them in some other way.

Step 2: Set Up Parsing of Console Output

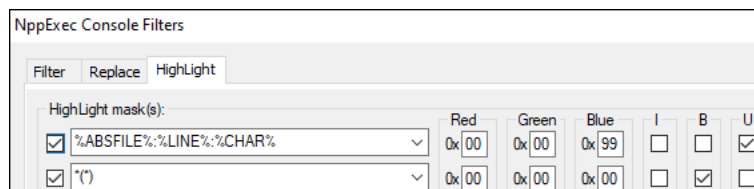
The NppExec plugin allows you to parse console output and navigate to the appropriate line of code. You can also optionally apply specific formatting to the console output.

For instance, your console output can look like the following:

```
C:\MyProj\myFile.c:12:8:
  Useless if (DEFECT:USELESS_IF)
  Details: Unnecessary code, if-condition is always true.
C:\MyProj\myFile.c:15:20:
  Array access out of bounds (DEFECT:OUT_BOUND_ARRAY)
  Details: Attempt to access to array element 20.
  Valid index range: [0 .. 19].
```

You can set up the output so that clicking a link directly takes you to the start of the relevant token on the relevant line of code.

To set up this presentation of results, select **Plugins > NppExec** and then select the **Console Output Filters** option. The following options allow the previous presentation of results:



The first highlight mask indicates that lines having the format

```
.....
```

contain the absolute path to the file before the first colon, the line number between the first and second colon, and the column number (or character number) after the second colon. The mask reads the information (file, line and column), underlines these lines and colors them blue.

The second highlight mask simply bolds lines having the format

```
...(...)
```

These lines contain the result name, for instance, the name of a defect.

Further Exploration

The official Polyspace as You Code extensions enable other actions such as analyzing build commands, configuring checkers, and downloading baselines from the Polyspace Access web server. In a real development environment, you want to analyze your build commands to emulate your compilation toolchain as closely as possible, configure the checkers that are most meaningful to you, and baseline results so that you focus only on new results coming from your changes.

You can extend the approach described here to create menu items in your IDE or editor for all these actions. For more information on these workflows from the command line, see:

- “Generate Build Options for Polyspace as You Code Analysis at the Command Line” on page 11-41
- “Configure Checkers for Polyspace as You Code at the Command Line” on page 11-70
- “Baseline Polyspace as You Code Results on Command Line” on page 11-56

With your IDE or editor set up for Polyspace as You Code, you can create a quality gate for submission. You can set up a configuration with checkers for which you do not want any finding in your submission. Before submitting a file, you can make sure that you have fixed all findings from those checkers.

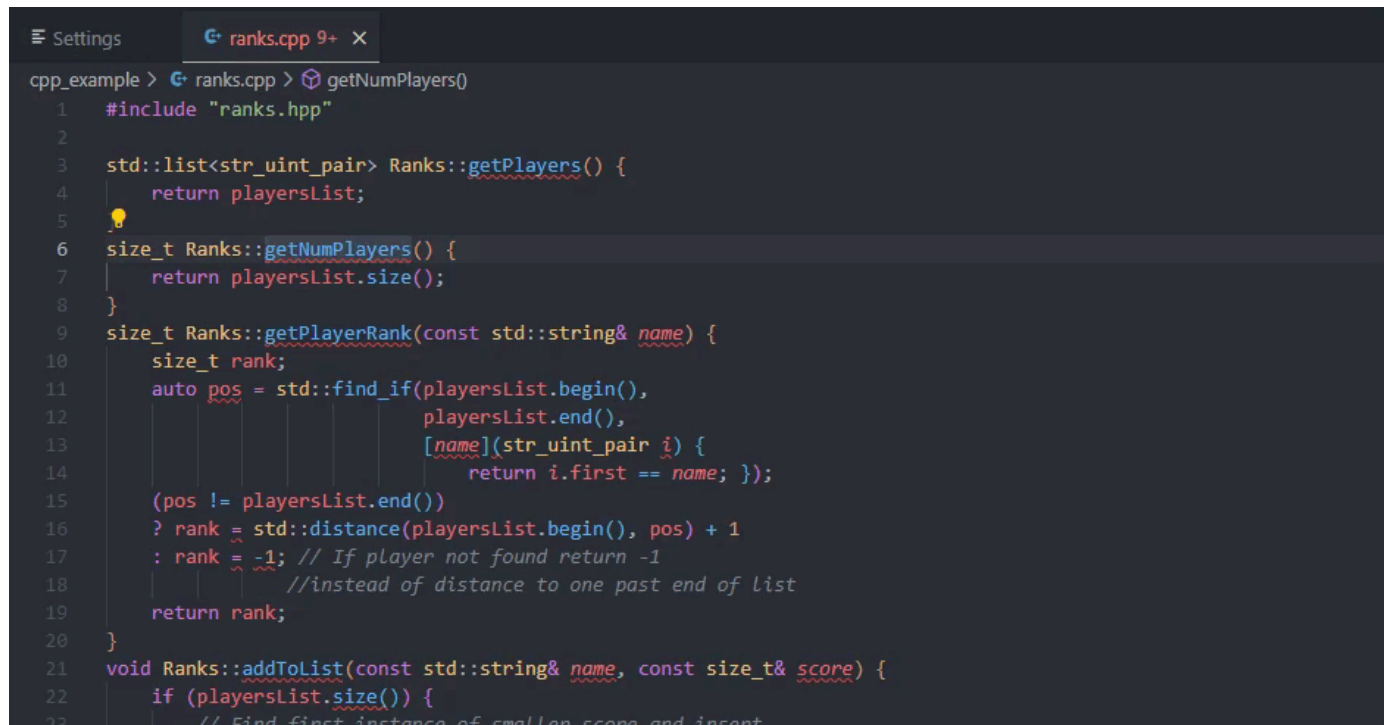
See Also

`polyspace-bug-finder-access | polyspace-results-export`

Use a Justification Catalog to Autocomplete Annotations in Polyspace as You Code plugins

After you run an analysis, you can suppress known or acceptable defects or coding rule deviations by adding code annotations to indicate that the issue has been reviewed and will not be fixed.

If your organization has a predefined set of comments that you use to justify known acceptable defects and coding standard deviations, you can store these comments in a justification catalog where you associate one or more comment with each defect or coding rule deviation. You can then pass this catalog to the Polyspace as You Code Visual Studio Code extension or Eclipse plugin and use the predefined comments to autocomplete code annotations comments when you annotate the code.



```

cpp_example > ranks.cpp > getNumPlayers()
1  #include "ranks.hpp"
2
3  std::list<str_uint_pair> Ranks::getPlayers() {
4      return playersList;
5
6  size_t Ranks::getNumPlayers() {
7      return playersList.size();
8  }
9  size_t Ranks::getPlayerRank(const std::string& name) {
10     size_t rank;
11     auto pos = std::find_if(playersList.begin(),
12                            playersList.end(),
13                            [name](str_uint_pair i) {
14                                return i.first == name; });
15     (pos != playersList.end())
16     ? rank = std::distance(playersList.begin(), pos) + 1
17     : rank = -1; // If player not found return -1
18     //instead of distance to one past end of list
19     return rank;
20 }
21 void Ranks::addToList(const std::string& name, const size_t& score) {
22     if (playersList.size()) {
23         // Find first instance of smaller score and insert

```

You pass the catalog to Polyspace as You Code by specifying the path of the catalog file in these settings:

- Visual Studio Code — `polyspace.justification.catalog` setting. See “Justification Catalog” on page 11-15.
- Eclipse — **Justification catalog** setting. See “Polyspace justification” on page 11-19.

Create and Edit Justification Catalog

The justification catalog is a JSON file that uses the Polyspace code annotation syntax. For details of the syntax, see “Annotation Syntax Details” on page 30-4. To create a catalog, copy `justification_catalog.json` to a text editor and save it as a JSON file with UTF-8 encoding.

`justification_catalog.json`

```
{
  "justifications": [
```

```

    {
      "family": "Defect",
      "acronym": "METHOD_NOT_CONST",
      "comment": [
        "This is my preset justification",
        "Another justification for same defect"
      ]
    },
    {
      "family": "*",
      "acronym": "INT_ZERO_DIV",
      "comment": "Family wildcard justification"
    },
    {
      "family": "Defect",
      "acronym": "*",
      "comment": "Rule wildcard justification applies to all defects"
    },
    {
      "family": "MISRA C:2012",
      "acronym": "21.*",
      "comment": "Justification for 21.* rules"
    },
    {
      "family": "AUTOSAR-CPP14",
      "acronym": "M9-3-3",
      "comment": "Preset justification for M9-3-3"
    }
  ]
}

```

The JSON structure consists of a `justifications` array which contains a collection of objects with at least three key-value pairs each. An object represents a result and the comments that you want to associate with that result.

```

{
  "justifications": [
    {
      "family": "familyValue",
      "acronym": "acronymValue",
      "comment": [
        "autocomplete comment for family:acronym result",
        "some other autocomplete comment for the same result"
      ]
    }
  ]
}

```

- Specify a Polyspace results family value for the `family` key.
- Specify a Polyspace result name value for the `acronym` key.
- Specify a comment for `comment` key to associate that comment with the Polyspace results with the family and result name that you specified in the other key-value pairs.

To associate additional comments with a result, add more comment key-value pairs in the object the represents that finding.

For example, in the file `justification_catalog.json`, the comments "This is my preset justification" and "Another justification for same defect" are available for autocomplete when you type an annotation for defect `METHOD_NOT_CONST`. Because the comment "Rule wildcard justification applies to all defects" applies to all defects, this comment is also available for autocomplete when you annotate `METHOD_NOT_CONST`.

See Also

External Websites

- “Annotate Code and Hide Known or Acceptable Results” on page 30-2

Review Workflows Common to All Platforms

Hide Known or Acceptable Results Using Code Annotations

Annotate Code and Hide Known or Acceptable Results

If a Polyspace analysis of your code finds known or acceptable defects or coding rule violations, you can suppress the defects or violations in subsequent analyses. Add code annotations indicating that you have reviewed the issues and that you do not intend to fix them.

You can add annotations through menu items in the Polyspace user interface (or IDE plugins) or by typing them directly in your code. For the general workflow of adding annotations:

- In the Polyspace desktop user interface, see “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 22-2.
- In the Polyspace Access web interface, see “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 26-2.
- In IDEs using Polyspace as You Code plugins or extensions, see “Review Polyspace as You Code Results in IDEs”.

This topic shows the annotation syntax.

Note that you cannot hide the run-time errors detected with Code Prover from your source code even with code annotations. However, like all other results, the review information associated with a run-time error is extracted from the corresponding code annotation and shown with the result.

Code Annotation Syntax

To add comments directly to your source file, use the Polyspace annotation syntax. The syntax is not case sensitive, and has the following format. Both C style comments within `/* */` and C++ style comments starting with `//` are supported.

Annotating Single Line of Code

To annotate a result on the current line of code (including macros), use this syntax:

```
line of code; /* polyspace Family:Result_name */
```

For instance:

```
var++; /* polyspace DEFECT:INT_OVFL */
```

Annotations begin with the keyword `polyspace` and must include the *Family* and *Result_name* field values.

You can optionally specify a *Status*, *Severity*, and *Comment* field value:

```
polyspace Family:Result_name [Status:Severity] "Comment"
```

For instance:

```
var++; /* polyspace DEFECT:INT_OVFL [Justified:Low] "Overflow taken into
account."*/
```

If you do not specify a status, Polyspace considers the result justified, and assigns the status `No action planned` to the result.

For further details, see “Annotation Syntax Details” on page 30-4 and “Syntax Examples” on page 30-6.

Annotating Code Block

To annotate a block of code, use the following syntax. Note that the annotations apply only to the block of code itself and not to bodies of functions called from the block.

- Annotation for current line of code and *n* following lines:

```
line of code; /* polyspace +n Family:Result_name */
```

- Annotation for block of code:

```
/* polyspace-begin Family:Result_name */
{
    block of code
}
/* polyspace-end Family:Result_name */
```

Optionally, specify a status, severity and comment.

If annotations for results with the same *Family* and *Result_name* are nested, the innermost annotation is used.

For example, in this code, the annotation on line 9 is applied instead of the block annotation, but the block annotation is applied to the violation on line 7.

```
1 /*polyspace-begin MISRA-C:14.9 [To fix:High] */
2 int main(void)
3 {
4     int x = 1;
5     int y = x / 2;
6
7     if (y < 0) /* Block annotation is applied to this violation of MISRA-C:14.9*/
8         y++;
9     if (x > y) /*polyspace MISRA-C:14.9 [Justified:Low] */
10        return x;
11    return x;
12 }
13 /*polyspace-end MISRA-C:14.9 [To fix:High] "Block annotation"*/
```

When you annotate a code block, the annotation applies only to the issues that arise from within the block. For instance, say you have a function call in the annotated block, and the body of the function gives rise to a violation. This violation is not affected by the annotation around the code block where the function is called.

For further details, see “Annotation Syntax Details” on page 30-4 and “Syntax Examples” on page 30-6.

Justifying Multiple Results in One Annotation

To justify multiple results in the same annotation, use the following syntax.

- If the results belong to the same family, specify comma-separated result names.

```
line of code; /* polyspace Family:Result_1_name,Result_2_name */
```

- If the results belong to different families, specify space-separated family names.

```
line of code; /* polyspace Family_1:Result_1_name Family_2:Result_2_name */
```

Optionally, specify a status, severity and comment.

For further details, see “Annotation Syntax Details” on page 30-4 and “Syntax Examples” on page 30-6.

Annotation Syntax Details

To replace the different annotation fields with their allowed values, use the values in this table or see the examples on page 30-6.

Field	Allowed Value
<i>Family</i>	<p>Type of analysis result:</p> <ul style="list-style-type: none"> • DEFECT (Polyspace Bug Finder) • RTE, for run-time checks (Polyspace Code Prover) • CODE-METRICS, for function-level code complexity metrics • VARIABLE, for global variables (Polyspace Code Prover) • MISRA-C or MISRA2004 for MISRA C: 2004 rule violations. These annotations also apply to MISRA C: 2012 violations based on the mapping between the two standards. The mapping allows you to reuse your justifications for the older standard when migrating to the newer one. See “Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results” on page 20-5. • MISRA-AC-AGC for violations of MISRA C:2004 rules applicable to generated code • MISRA-C3 or MISRA2012 for MISRA C: 2012 rule violations. The annotation works even for the rules applicable to generated code. • CERT-C for CERT C coding standard violations • CERT-CPP for CERT C++ coding standard violations • ISO-17961 for ISO/IEC TS 17961 coding standard violations • MISRA-CPP for MISRA C++ rule violations • AUTOSAR-CPP14 for AUTOSAR C++14 rule violations • JSF for JSF++ rule violations • GUIDELINE for software complexity guidelines. • CUSTOM for violations of custom coding rules <p>To specify all analysis results, use the asterisk character * : *.</p> <p>See “Syntax Examples” on page 30-28.</p>

Field	Allowed Value
<i>Result_name</i>	<p>For DEFECT, use short names of checkers. See “Short Names of Bug Finder Defect Groups and Defect Checkers” on page 30-11.</p> <p>For RTE, use short names of run-time checks. See “Short Names of Code Prover Run-Time Checks” (Polyspace Code Prover).</p> <p>For CODE-METRICS, use short names of code complexity metrics. See “Short Names of Code Complexity Metrics” on page 30-24. Note that project and file metrics cannot be justified using code annotations.</p> <p>For VARIABLE, the only allowed value is the asterisk character " * ".</p> <p>For coding standard violations, specify the rule number or numbers.</p> <p>For software complexity guidelines, use acronyms for the guidelines. See pages for individual guidelines in “Guidelines”.</p> <p>To specify all parts of a result name [MISRA2012:17.*] or all result names in a family [DEFECT:*], use the asterisk character.</p> <p>See “Syntax Examples” on page 30-28.</p>
<i>Status</i>	<p>Text to indicate how you intend to address the error in your code. This value populates the Status column in the Results List pane as:</p> <ul style="list-style-type: none"> • Unreviewed • To investigate • To fix • Justified • No action planned • Not a defect • Other <p>Polyspace suppresses results annotated with status Justified, No action planned, or Not a defect in subsequent analyses. If you specify a status that is not an allowed value, Polyspace stores it as a custom status.</p>

Field	Allowed Value
<i>Severity</i>	<p>Text to indicate how critical you consider the error in your code. This value populates the Severity column in the Results List pane as:</p> <ul style="list-style-type: none"> • Unset • High • Medium • Low <p>If you specify a severity that is not an allowed value, Polyspace appends it to the status field and stores it as a custom status. For example, [To investigate:sporadic] is displayed in the Status column of the Results List pane as To investigate sporadic.</p>
<i>Comment</i>	<p>Additional text, such as a keyword or an explanation for the status and severity. This value populates the Comment column in the Results List pane.</p> <p>The additional text can span more than one line in the code. When showing this text in reports, leading and trailing spaces on a line are merged into one space so that the entire text can be read as a single paragraph.</p>

Syntax Examples

Suppress a Single Defect

Enter an annotation on the same line as the defect and specify the *Family* (DEFECT) and the *Result_name* (INT_OVFL). When you do not specify a status, Polyspace assigns the status No action planned, and then suppresses the result in subsequent analyses.

```
int var = INT_MAX;
var++; /* polyspace DEFECT:INT_OVFL */
```

Suppress a Single Coding Standard Violation

Justify a coding standard violation, for instance, a CERT-C violation.

Enter an annotation on the same line as the violation and specify the *Family* (CERT-C) and the *Result_name* (the rule number, for instance, STR31-C). Assign the status Justified, severity Low and a comment.

```
line of code; /* polyspace CERT-C:STR31-C [Justified:Low] "Overflow cannot happen
because of external constraints." */
```


Suppress All MISRA C: 2012 Violations Over Multiple Lines

Enter an annotation with `+n` between `polyspace` and the `Family:Result_name` entries. The annotation applies to the same line and the `n` following lines.

This annotation applies to lines 4-7. The line count includes code, comments, and blank lines.

```
4. line of code ; // polyspace +3 MISRA2012:*
5. //comment
6.
7. line of code;
8. line of code;
```

Suppress All Code Metrics on Function

To annotate function-level code complexity metrics, in the function definition, enter an annotation on the same line as the function name.

This annotation suppresses all code complexity metrics for function `func`:

```
char func(char param) { //polyspace CODE-METRICS:*
    ...
}
```

Specify Multiple Families in the Same Annotation

Enter each family separated by a space. This annotation applies to all MISRA C:2012 rules 17 and to all run-time checks.

```
line of code; /* polyspace MISRA2012:17.* RTE:* */
```

Specify Multiple Result Names in the Same Annotation

After you specify the *Family* (DEFECT), enter each *Result_name* separated by a comma.

```
system("rm ~/.config"); /* polyspace DEFECT:UNSAFE_SYSTEM_CALL,RETURN_NOT_CHECKED */
```

Suppress Result Showing Global Variable Usage

To justify a Code Prover result showing global variable usage, for instance, an unused global variable, enter the annotation next to the variable declaration.

For instance, to suppress a global variable result with a *Justified* status, *Low* severity and some comments, you can enter an annotation like this:

```
int var; /* polyspace VARIABLE:* [Justified: Low] "Storage repo for later use"*/
```

Add Explanatory Notes to Annotation

After you specify a *Family* and a *Result_name*, you can add a *Comment* with additional information for your justification. You can provide a comment for all families and result names, or a comment for each family or result name.

```
//Single comment
line of code; /* polyspace DEFECT:BAD_FREE MISRA2004:* "OK Defect and MISRA" */
//Multiple comments incorrect syntax:
line of code; /* polyspace DEFECT:* "OK defect" MISRA2004:5.2 "OK MISRA" */
//Multiple comments correct syntax:
line of code; /* polyspace DEFECT:* "OK defect" polyspace MISRA2004:5.2 "OK MISRA" */
```

In annotations, Polyspace ignores all text following double quotes. To specify additional *Family:Result_name*, *[Status:Severity]* or *Comment* entries, you must reenter the keyword `polyspace` after text in double quotes.

Set Status and Severity

You can specify allowed values on page 30-2 or enter custom values for status and severity. A custom severity entry is appended to the status and stored as a custom **Status** in the user interface.

```
//Set Status only
line of code; /* polyspace DEFECT:* [To fix] "some comment" */

//Set Status 'To fix' and Severity 'High'
line of code; /* polyspace VARIABLE:* [To fix: High] "some comment"*/

//Set custom status 'Assigned' and Severity 'Medium'
line of code; /* polyspace MISRA2012:12.* [Assigned: Medium] */
```

Justify Violations in a Code Block

Use annotation to justify violations arising from a block of code. For instance, consider this code:

```
double foo(void){
    constexpr int speedLimit = 65;
    constexpr double coeff = 0.2;
    int flag{0};
    int negOne{-1};
    //...
    return (flag)?speedLimit*coeff*negOne
    : speedLimit*coeff*negOne - 35; //Noncompliant
}

int main(){

    /* polyspace-begin AUTOSAR-CPP14:A5-1-1 [Justified: Low]"Known Constant"*/
    //....
    for(int i = 0; i<10;++i){
        foo();
        //...
    }

    /* polyspace-end AUTOSAR-CPP14:A5-1-1 [Justified: Low]"Known Constant"*/
    return 1;
}
```

The `for` loop has hard coded literal as the loop boundary, which is a violation of AUTOSAR C++14 Rule A5-1-1. By annotating the block with the syntax `/*polyspace begin...*/ {...}/` `*polyspace-end...*/`, the violation in the line `for(int i = 0; i<10;++i){` appears as a justified defect in the results list.

The annotated block contains a call to the function `foo()`. The annotations do not apply to the body of this function. For instance, the violation in the return statement of `foo()` appears as an unreviewed defect.

Code Annotation Warnings

If you enter a code annotation incorrectly or the annotation no longer applies, the analysis log contains a warning:

Warning: These Polyspace annotations do not apply to the current code

The warning can mean one of the following:

- The issue is no longer detected because of code fixes or changes in analysis configuration.

For instance, an annotation such as:

```
/* polyspace RTE:IDP [No action planned:Low] */
```

Might no longer apply because the **Illegally dereferenced pointer** check (annotated as IDP), which was previously red or orange, is now green.

- The annotation syntax is incorrect.

An annotation beginning with `polyspace` followed by a word and then a `:` (colon) such as:

```
// polyspace Family :
```

is considered as a Polyspace annotation justifying a result. If the word *Family* following `polyspace` is not a type of Polyspace result such as DEFECT or RTE, the analysis considers the annotation as invalid and shows the warning. For instance, this annotation triggers the warning:

```
// polyspace TODO: Fix in March dev cycle
```

since `TODO` is not a type of Polyspace result. To avoid these warnings, use another separator, for instance, instead of a colon. For the full list of result types, see “Code Annotation Syntax” on page 30-2.

Ignoring Code Annotations

In some cases, you might want to run a clean analysis as if the results have not been previously reviewed. For instance, you might want to perform a worst-case analysis where you see all previously justified results.

You can use the option `-ignore-code-annotations` to run such an analysis with no history. The analysis ignores the code annotations and shows all annotated results without any review information taken from the annotations.

See also `-ignore-code-annotations`.

See Also

`-xml-annotations-description` | `-ignore-code-annotations`

More About

- “Define Custom Annotation Format” on page 30-30
- “Short Names of Bug Finder Defect Groups and Defect Checkers” on page 30-11
- “Short Names of Code Complexity Metrics” on page 30-24

Short Names of Bug Finder Defect Groups and Defect Checkers

To justify defects through code annotations, use the checker command-line names, or short names, listed in “Bug Finder Defect Checkers Short Names” on page 30-11.

You can also enable the detection of a specific defect by using its checker short name as argument of the `-checkers` option. Instead of listing individual defects, specify groups of defects by the group name, for instance, `numerical`, `data_flow`, and so on.

See `Find defects (-checkers)` and “Bug Finder Defect Groups” on page 18-43.

Bug Finder Defect Groups Short Names

Defect Group	Command-Line Parameter
“C++ Exception Defects”	<code>cpp_exceptions</code>
“Concurrency Defects”	<code>concurrency</code>
“Cryptography Defects”	<code>cryptography</code>
“Data Flow Defects”	<code>data_flow</code>
“Dynamic Memory Defects”	<code>dynamic_memory</code>
“Good Practice Defects”	<code>good_practice</code>
“Numerical Defects”	<code>numerical</code>
“Object Oriented Defects”	<code>object_oriented</code>
“Performance Defects”	<code>performance</code>
“Programming Defects”	<code>programming</code>
“Resource Management Defects”	<code>resource_management</code>
“Static Memory Defects”	<code>static_memory</code>
“Security Defects”	<code>security</code>
“Tainted Data Defects”	<code>tainted_data</code>

Bug Finder Defect Checkers Short Names

Defect	Command-line Name
<code>*this</code> not returned in copy assignment operator	<code>RETURN_NOT_REF_TO_THIS</code>
Abnormal termination of exit handler	<code>EXIT_ABNORMAL_HANDLER</code>
Absorption of float operand	<code>FLOAT_ABSORPTION</code>
Accessing object with temporary lifetime	<code>TEMP_OBJECT_ACCESS</code>
Alignment changed after memory reallocation	<code>ALIGNMENT_CHANGE</code>
Alternating input and output from a stream without flush or positioning call	<code>IO_INTERLEAVING</code>
Ambiguous declaration syntax	<code>MOST_VEXING_PARSE</code>
A move operation may throw	<code>MOVE_OPERATION_MAY_THROW</code>

Defect	Command-line Name
Arithmetic operation with NULL pointer	NULL_PTR_ARITH
Array access out of bounds	OUT_BOUND_ARRAY
Array access with tainted index	TAINTED_ARRAY_INDEX
Assertion	ASSERT
Asynchronously cancellable thread	ASYNCHRONOUSLY_CANCELLABLE_THREAD
Atomic load and store sequence not atomic	ATOMIC_VAR_SEQUENCE_NOT_ATOMIC
Atomic variable accessed twice in an expression	ATOMIC_VAR_ACCESS_TWICE
Automatic or thread local variable escaping from a thread	LOCAL_ADDR_ESCAPE_THREAD
Bad file access mode or status	BAD_FILE_ACCESS_MODE_STATUS
Bad order of dropping privileges	BAD_PRIVILEGE_DROP_ORDER
Base class assignment operator not called	MISSING_BASE_ASSIGN_OP_CALL
Base class destructor not virtual	DTOR_NOT_VIRTUAL
Bitwise and arithmetic operation on the same data	BITWISE_ARITH_MIX
Bitwise operation on negative value	BITWISE_NEG
Blocking operation while holding lock	BLOCKING_WHILE_LOCKED
Buffer overflow from incorrect string format specifier	STR_FORMAT_BUFFER_OVERFLOW
Bytewise operations on nontrivial class object	MEMOP_ON_NONTRIVIAL_OBJ
C++ reference to const-qualified type with subsequent modification	WRITE_REFERENCE_TO_CONST_TYPE
C++ reference type qualified with const or volatile	CV_QUALIFIED_REFERENCE_TYPE
Call through non-prototyped function pointer	UNPROTOTYPED_FUNC_CALL
Call to memset with unintended value	MEMSET_INVALID_VALUE
Character value absorbed into EOF	CHAR_EOF_CONFUSED
Closing a previously closed resource	DOUBLE_RESOURCE_CLOSE
Code deactivated by constant false condition	DEACTIVATED_CODE
Command executed from externally controlled path	TAINTED_PATH_CMD
Constant block cipher initialization vector	CRYPTO_CIPHER_CONSTANT_IV
Constant cipher key	CRYPTO_CIPHER_CONSTANT_KEY
Const parameter values may cause unnecessary data copies	CONST_PARAMETER_VALUE

Defect	Command-line Name
Const return values may cause unnecessary data copies	CONST_RETURN_VALUE
Const rvalue reference parameter may cause unnecessary data copies	CONST_RVALUE_REFERENCE_PARAMETER
Const std::move input may cause a more expensive object copy	EXPENSIVE_STD_MOVE_CONST_OBJECT
Context initialized incorrectly for cryptographic operation	CRYPTO_PKEY_INCORRECT_INIT
Context initialized incorrectly for digest operation	CRYPTO_MD_BAD_FUNCTION
Conversion or deletion of incomplete class pointer	INCOMPLETE_CLASS_PTR
Copy constructor not called in initialization list	MISSING_COPY_CTOR_CALL
Copy of overlapping memory	OVERLAPPING_COPY
Copy operation modifying source operand	COPY_MODIFYING_SOURCE
Critical data member is not private	CRITICAL_DATA_MEMBER_DECLARED_PUBLIC
C string from string::c_str() compared to pointer	STD_STRING_C_STR_COMPARED_TO_POINTER
Data race	DATA_RACE
Data race on adjacent bit fields	DATA_RACE_BIT_FIELDS
Data race through standard library function call	DATA_RACE_STD_LIB
Dead code	DEAD_CODE
Deadlock	DEADLOCK
Deallocation of previously deallocated pointer	DOUBLE_DEALLOCATION
Declaration mismatch	DECL_MISMATCH
Declaration of catch for generic exception	CATCH_FOR_GENERIC_EXCEPTION
Declaration of throw for generic exception	THROW_FOR_GENERIC_EXCEPTION
Delete of void pointer	DELETE_OF_VOID_PTR
Destination buffer overflow in string manipulation	STRLIB_BUFFER_OVERFLOW
Destination buffer underflow in string manipulation	STRLIB_BUFFER_UNDERFLOW
Destruction of locked mutex	DESTROY_LOCKED
Deterministic random output from constant seed	RAND_SEED_CONSTANT
Double lock	DOUBLE_LOCK
Double unlock	DOUBLE_UNLOCK
Duplicated code	DUPLICATED_CODE
Empty destructors may cause unnecessary data copies	EMPTY_DESTRUCTOR_DEFINED

Defect	Command-line Name
Environment pointer invalidated by previous operation	INVALID_ENV_POINTER
Errno not checked	ERRNO_NOT_CHECKED
Errno not reset	MISSING_ERRNO_RESET
Exception caught by value	EXCP_CAUGHT_BY_VALUE
Exception handler hidden by previous handler	EXCP_HANDLER_HIDDEN
Execution of a binary from a relative path can be controlled by an external actor	RELATIVE_PATH_CMD
Execution of externally controlled command	TAINTED_EXTERNAL_CMD
Expensive allocation in loop	EXPENSIVE_ALLOC_IN_LOOP
Expensive constant <code>std::string</code> construction	EXPENSIVE_CONSTANT_STD_STRING
Expensive copy in a range-based for loop iteration	EXPENSIVE_RANGE_BASED_FOR_LOOP_ITERATION
Expensive dynamic cast	EXPENSIVE_DYNAMIC_CAST
Expensive local variable copy	EXPENSIVE_LOCAL_VARIABLE
Expensive logical operation	EXPENSIVE_LOGICAL_OPERATION
Expensive pass by value	EXPENSIVE_PASS_BY_VALUE
Expensive post-increment operation	EXPENSIVE_POST_INCREMENT
Expensive return by value	EXPENSIVE_RETURN_BY_VALUE
Expensive return caused by unnecessary <code>std::move</code>	EXPENSIVE_RETURN_STD_MOVE
Expensive return of a <code>const</code> object	EXPENSIVE_RETURN_CONST_OBJECT
Expensive use of a standard algorithm when a more efficient method exists	EXPENSIVE_USE_OF_STD_ALGORITHM
Expensive use of container's <code>count</code> method	EXPENSIVE_CONTAINER_COUNT
Expensive use of container's <code>insertion</code> method	EXPENSIVE_CONTAINER_INSERTION
Expensive use of container's <code>size</code> method	EXPENSIVE_CONTAINER_EMPTYNESS_CHECK
Expensive use of <code>map</code> 's bracket operator to insert or assign a value	EXPENSIVE_MAP_INSERT_OR_ASSIGN
Expensive use of non-member <code>std::string</code> <code>operator+()</code> instead of a simple <code>append</code>	EXPENSIVE_STD_STRING_APPEND
Expensive use of <code>std::string</code> methods instead of more efficient overload	EXPENSIVE_USE_OF_STD_STRING_METHODS

Defect	Command-line Name
Expensive use of std::string with empty string literal	UNNECESSARY_EMPTY_STRING_LITERAL
Expensive use of string functions from the C standard library	EXPENSIVE_USE_OF_C_STRING_API
Expensive use of substr() to shorten a std::string	EXPENSIVE_STD_STRING_RESIZE
File access between time of check and use (TOCTOU)	TOCTOU
File descriptor exposure to child process	FILE_EXPOSURE_TO_CHILD
File does not compile	file_does_not_compile
File manipulation after chroot() without chdir("/")	CHROOT_MISUSE
Float conversion overflow	FLOAT_CONV_OVFL
Float division by zero	FLOAT_ZERO_DIV
Floating point comparison with equality operators	BAD_FLOAT_OP
Float overflow	FLOAT_OVFL
Format string specifiers and arguments mismatch	STRING_FORMAT
Function called from signal handler not asynchronous-safe	SIG_HANDLER_ASYNC_UNSAFE
Function called from signal handler not asynchronous-safe (strict)	SIG_HANDLER_ASYNC_UNSAFE_STRICT
Function pointer assigned with absolute address	FUNC_PTR_ABSOLUTE_ADDR
Function that can spuriously fail not wrapped in loop	SPURIOUS_FAILURE_NOT_WRAPPED_IN_LOOP
Function that can spuriously wake up not wrapped in loop	SPURIOUS_WAKEUP_NOT_WRAPPED_IN_LOOP
Hard-coded buffer size	HARD_CODED_BUFFER_SIZE
Hard-coded loop boundary	HARD_CODED_LOOP_BOUNDARY
Hard-coded object size used to manipulate memory	HARD_CODED_MEM_SIZE
Hard-coded sensitive data	HARD_CODED_SENSITIVE_DATA
Host change using externally controlled elements	TAINTED_HOSTID
Improper array initialization	IMPROPER_ARRAY_INIT
Improper erase-remove idiom	STD_REMOVE_WITHOUT_ERASE
Inappropriate I/O operation on device files	INAPPROPRIATE_IO_ON_DEVICE
Incompatible padding for RSA algorithm operation	CRYPTO_RSA_BAD_PADDING
Incompatible types prevent overriding	VIRTUAL_FUNC_HIDING
Inconsistent cipher operations	CRYPTO_CIPHER_BAD_FUNCTION
Incorrect data type passed to va_arg	VA_ARG_INCORRECT_TYPE

Defect	Command-line Name
Incorrect key for cryptographic algorithm	CRYPTO_PKEY_INCORRECT_KEY
Incorrectly indented statement	INCORRECT_INDENTATION
Incorrect order of network connection operations	BAD_NETWORK_CONNECT_ORDER
Incorrect pointer scaling	BAD_PTR_SCALING
Incorrect syntax of flexible array member size	FLEXIBLE_ARRAY_MEMBER_INCORRECT_SIZE
Incorrect type data passed to va_start	VA_START_INCORRECT_TYPE
Incorrect use of offsetof in C++	OFFSETOF_MISUSE
Incorrect use of va_start	VA_START_MISUSE
Incorrect value forwarding	INCORRECT_VALUE_FORWARDING
Inefficient string length computation	INEFFICIENT_BASIC_STRING_LENGTH
Inefficient use of for loop	PREFER_RANGE_BASED_FOR_LOOPS
Inefficient use of sprintf	inefficient_sprintf
Infinite loop	INFINITE_LOOP
Information leak via structure padding	PADDING_INFO_LEAK
Inline constraint not respected	INLINE_CONSTRAINT_NOT_RESPECTED
Integer constant overflow	INT_CONSTANT_OVFL
Integer conversion overflow	INT_CONV_OVFL
Integer division by zero	INT_ZERO_DIV
Integer overflow	INT_OVFL
Integer precision exceeded	INT_PRECISION_EXCEEDED
Invalid assumptions about memory organization	INVALID_MEMORY_ASSUMPTION
Invalid deletion of pointer	BAD_DELETE
Invalid file position	INVALID_FILE_POS
Invalid free of pointer	BAD_FREE
Invalid iterator usage	INVALID_ITERATOR_USAGE
Invalid scientific notation format	INVALID_NOTATION_ON_E_CONSTANT
Invalid use of == operator	BAD_EQUAL_EQUAL_USE
Invalid use of = operator	BAD_EQUAL_USE
Invalid use of standard library floating point routine	FLOAT_STD_LIB

Defect	Command-line Name
Invalid use of standard library integer routine	INT_STD_LIB
Invalid use of standard library memory routine	MEM_STD_LIB
Invalid use of standard library routine	OTHER_STD_LIB
Invalid use of standard library string routine	STR_STD_LIB
Invalid va_list argument	INVALID_VA_LIST_ARG
Join or detach of a joined or detached thread	DOUBLE_JOIN_OR_DETACH
Lambda used as typeid operand	LAMBDA_TYPE_MISUSE
LDAP injection	LDAP_INJECTION
Library loaded from externally controlled path	TAINED_PATH_LIB
Line with more than one statement	MORE_THAN_ONE_STATEMENT
Load of library from a relative path can be controlled by an external actor	RELATIVE_PATH_LIB
Loop bounded with tainted value	TAINED_LOOP_BOUNDARY
Macro terminated with a semicolon	SEMICOLON_TERMINATED_MACRO
Macro with multiple statements	MULTI_STMT_MACRO
Member not initialized in constructor	NON_INIT_MEMBER
Memory allocation with tainted size	TAINED_MEMORY_ALLOC_SIZE
Memory comparison of float-point values	MEMCMP_FLOAT
Memory comparison of padding data	MEMCMP_PADDING_DATA
Memory comparison of strings	MEMCMP_STRINGS
Memory leak	MEM_LEAK
Method not const	METHOD_NOT_CONST
Mismatch between data length and size	DATA_LENGTH_MISMATCH
Mismatched alloc/dealloc functions on Windows	WIN_MISMATCH_DEALLOC
Missing blinding for RSA algorithm	CRYPTO_RSA_NO_BLINDING
Missing block cipher initialization vector	CRYPTO_CIPHER_NO_IV
Missing break of switch case	MISSING_SWITCH_BREAK
Missing byte reordering when transferring data	MISSING_BYTESWAP
Missing call to container's reserve method	MISSING_CONTAINER_RESERVE
Missing case for switch condition	MISSING_SWITCH_CASE
Missing certification authority list	CRYPTO_SSL_NO_CA
Missing cipher algorithm	CRYPTO_CIPHER_NO_ALGORITHM
Missing cipher data to process	CRYPTO_CIPHER_NO_DATA
Missing cipher final step	CRYPTO_CIPHER_NO_FINAL

Defect	Command-line Name
Missing cipher key	CRYPTO_CIPHER_NO_KEY
Missing constexpr specifier	MISSING_CONSTEXPR
Missing data for encryption, decryption or signing operation	CRYPTO_PKEY_NO_DATA
Missing explicit keyword	MISSING_EXPLICIT_KEYWORD
Missing final step after hashing update operation	CRYPTO_MD_NO_FINAL
Missing hash algorithm	CRYPTO_MD_NO_ALGORITHM
Missing lock	BAD_UNLOCK
Missing null in string array	MISSING_NULL_CHAR
Missing or double initialization of thread attribute	BAD_THREAD_ATTRIBUTE
Missing overload of allocation or deallocation function	MISSING_OVERLOAD_NEW_DELETE_PAIR
Missing padding for RSA algorithm	CRYPTO_RSA_NO_PADDING
Missing parameters for key generation	CRYPTO_PKEY_NO_PARAMS
Missing peer key	CRYPTO_PKEY_NO_PEER
Missing private key	CRYPTO_PKEY_NO_PRIVATE_KEY
Missing private key for X.509 certificate	CRYPTO_SSL_NO_PRIVATE_KEY
Missing public key	CRYPTO_PKEY_NO_PUBLIC_KEY
Missing reset of a freed pointer	MISSING_FREED_PTR_RESET
Missing return statement	MISSING_RETURN
Missing salt for hashing operation	CRYPTO_MD_NO_SALT
Missing unlock	BAD_LOCK
Missing virtual inheritance	MISSING_VIRTUAL_INHERITANCE
Missing X.509 certificate	CRYPTO_SSL_NO_CERTIFICATE
Misuse of a FILE object	FILE_OBJECT_MISUSE
Misuse of errno	ERRNO_MISUSE
Misuse of errno in a signal handler	SIG_HANDLER_ERRNO_MISUSE
Misuse of narrow or wide character string	NARROW_WIDE_STR_MISUSE
Misuse of readlink()	READLINK_MISUSE
Misuse of return value from nonreentrant standard function	NON_REENTRANT_STD_RETURN
Misuse of sign-extended character value	CHARACTER_MISUSE

Defect	Command-line Name
Misuse of structure with flexible array member	FLEXIBLE_ARRAY_MEMBER_STRUCTURE_MISUSE
Modification of internal buffer returned from nonreentrant standard function	WRITE_INTERNAL_BUFFER_RETURNED_FROM_STD_FUNC
Move operation on const object	MOVE_CONST_OBJECT
Move operation uses copy	MOVE_OPERATION_USES_COPY
Multiple mutexes used with same condition variable	MULTI_MUTEX_WITH_ONE_COND_VAR
Multiple threads waiting on same condition variable	SIGNALLED_COND_VAR_NOT_UNIQUE
No data added into context	CRYPTO_MD_NO_DATA
Noexcept function exits with exception	NOEXCEPT_FUNCTION_THROWS
Non-compliance with AUTOSAR specification	autosar_lib_non_compliance
Non-initialized pointer	NON_INIT_PTR
Non-initialized variable	NON_INIT_VAR
Nonsecure hash algorithm	CRYPTO_MD_WEAK_HASH
Nonsecure parameters for key generation	CRYPTO_PKEY_WEAK_PARAMS
Nonsecure RSA public exponent	CRYPTO_RSA_LOW_EXPONENT
Nonsecure SSL/TLS protocol	CRYPTO_SSL_WEAK_PROTOCOL
Null pointer	NULL_PTR
Object slicing	OBJECT_SLICING
Opening previously opened resource	DOUBLE_RESOURCE_OPEN
Operator new not overloaded for possibly overaligned class	MISSING_OVERLOAD_NEW_FOR_ALIGNED_OBJ
Overlapping assignment	OVERLAPPING_ASSIGN
Partially accessed array	PARTIALLY_ACCESSED_ARRAY
Partial override of overloaded virtual functions	PARTIAL_OVERRIDE
Partially duplicated code	ALMOST_DUPLICATED_CODE
Pointer access out of bounds	OUT_BOUND_PTR
Pointer dereference with tainted offset	TAINTED_PTR_OFFSET
Pointer or reference to stack variable leaving scope	LOCAL_ADDR_ESCAPE
Pointer to non-initialized value converted to const pointer	NON_INIT_PTR_CONV
Possible copy-paste error	COPY_PASTE_ERROR
Possible invalid operation on boolean operand	INVALID_OPERATION_ON_BOOLEAN
Possible misuse of sizeof	sizeof_MISUSE

Defect	Command-line Name
Possibly inappropriate data type for switch expression	INAPPROPRIATE_TYPE_IN_SWITCH
Possibly unintended evaluation of expression because of operator precedence rules	OPERATOR_PRECEDENCE
Precision loss in integer to float conversion	INT_TO_FLOAT_PRECISION_LOSS
Predefined macro used as an object	MACRO_USED_AS_OBJECT
Predictable block cipher initialization vector	CRYPTO_CIPHER_PREDICTABLE_IV
Predictable cipher key	CRYPTO_CIPHER_PREDICTABLE_KEY
Predictable random output from predictable seed	RAND_SEED_PREDICTABLE
Preprocessor directive in macro argument	PRE_DIRECTIVE_MACRO_ARG
Privilege drop not verified	MISSING_PRIVILEGE_DROP_CHECK
Public static field is not const	PUBLIC_STATIC_FIELD_NOT_CONST
Qualifier removed in conversion	QUALIFIER_MISMATCH
Redundant expression in sizeof operand	SIZEOF_USELESS_OP
Reference to un-named temporary	LOCAL_REF_TO_UNNAMED_TEMPORARY
Resource leak	RESOURCE_LEAK
Returned value of a sensitive function not checked	RETURN_NOT_CHECKED
Return from computational exception signal handler	SIG_HANDLER_COMP_EXCP_RETURN
Return of non const handle to encapsulated data member	BREAKING_DATA_ENCAPSULATION
Self assignment not tested in operator	MISSING_SELF_ASSIGN_TEST
Semicolon on same line as if, for or while statement	SEMICOLON_CTRL_STMT_SAME_LINE
Sensitive data printed out	SENSITIVE_DATA_PRINT
Sensitive heap memory not cleared before release	SENSITIVE_HEAP_NOT_CLEARED
Server certificate common name not checked	CRYPTO_SSL_HOSTNAME_NOT_CHECKED
Shared data access within signal handler	SIG_HANDLER_SHARED_OBJECT
Shift of a negative value	SHIFT_NEG
Shift operation overflow	SHIFT_OVFL
Side effect in arguments to unsafe macro	SIDE_EFFECT_IN_UNSAFE_MACRO_ARG

Defect	Command-line Name
Side effect of expression ignored	SIDE_EFFECT_IGNORED
Signal call from within signal handler	SIG_HANDLER_CALLING_SIGNAL
Signal call in multithreaded program	SIGNAL_USE_IN_MULTITHREADED_PROGRAM
Sign change integer conversion overflow	SIGN_CHANGE
SQL injection	SQL_INJECTION
Standard function call with incorrect arguments	STD_FUNC_ARG_MISMATCH
Static uncalled function	UNCALLED_FUNC
std::endl may cause an unnecessary flush	STD_ENDL_USE
std::move called on an unmovable type	STD_MOVE_UNMOVABLE_TYPE
std::string_view initialized with dangling pointer	DANGLING_STRING_VIEW
Stream argument with possibly unintended side effects	STREAM_WITH_SIDE_EFFECT
Subtraction or comparison between pointers to different arrays	PTR_TO_DIFF_ARRAY
Tainted division operand	TAINTED_INT_DIVISION
Tainted modulo operand	TAINTED_INT_MOD
Tainted NULL or non-null-terminated string	TAINTED_STRING
Tainted sign change conversion	TAINTED_SIGN_CHANGE
Tainted size of variable length array	TAINTED_VLA_SIZE
Tainted string format	TAINTED_STRING_FORMAT
Thread-specific memory leak	THREAD_MEM_LEAK
Throw argument raises unexpected exception	THROW_ARGUMENT_EXPRESSION_THROWS
TLS/SSL connection method not set	CRYPTO_SSL_NO_ROLE
TLS/SSL connection method set incorrectly	CRYPTO_SSL_BAD_ROLE
Too many va_arg calls for current argument list	TOO_MANY_VA_ARG_CALLS
Typedef mismatch	TYPEDEF_MISMATCH
Umask used with chmod-style arguments	BAD_UMASK
Uncaught exception	UNCAUGHT_EXCEPTION
Uncaught exception	UNCAUGHT_EXCEPTION
Uncertain memory cleaning	UNCERTAIN_MEMORY_CLEANING
Uncleared sensitive data in stack	SENSITIVE_STACK_NOT_CLEARED
Universal character name from token concatenation	PRE_UCNAME_JOIN_TOKENS
Unmodified variable not const-qualified	UNMODIFIED_VAR_NOT_CONST

Defect	Command-line Name
Unnamed namespace in header file	UNNAMED_NAMESPACE_IN_HEADER
Unnecessary construction before reassignment	UNNECESSARY_CONSTRUCTION_BEFORE_ASSIGNMENT
Unnecessary implementation of a special member function	UNNECESSARY_IMPL_OF_SPECIAL_MEMBER_FUNCTION
Unnecessary padding	UNNECESSARY_STRUCT_PADDING
Unnecessary use of <code>std::string::c_str()</code> or equivalent string methods	EXPENSIVE_C_STR_STD_STRING_CONSTRUCTION
Unprotected dynamic memory allocation	UNPROTECTED_MEMORY_ALLOCATION
Unreachable code	UNREACHABLE
Unreliable cast of function pointer	FUNC_CAST
Unreliable cast of pointer	PTR_CAST
Unsafe call to a system function	UNSAFE_SYSTEM_CALL
Unsafe conversion between pointer and integer	BAD_INT_PTR_CAST
Unsafe conversion from string to numerical value	UNSAFE_STR_TO_NUMERIC
Unsafe standard encryption function	UNSAFE_STD_CRYPT
Unsafe standard function	UNSAFE_STD_FUNC
Unsigned integer constant overflow	UINT_CONSTANT_OVFL
Unsigned integer conversion overflow	UINT_CONV_OVFL
Unsigned integer overflow	UINT_OVFL
Unused parameter	UNUSED_PARAMETER
Useless if	USELESS_IF
Useless Include	USELESS_INCLUDE
Useless preprocessor conditional directive	USELESS_PREPROC_CONDITION
Use of a forbidden C/C++ keyword	FORBIDDEN_KEYWORD
Use of a forbidden function	FORBIDDEN_FUNC
Use of a forbidden macro	FORBIDDEN_MACRO
Use of automatic variable as <code>putenv</code> -family function argument	PUTENV_AUTO_VAR
Use of dangerous standard function	DANGEROUS_STD_FUNC
Use of externally controlled environment variable	TAINTED_ENV_VARIABLE
Use of indeterminate string	INDETERMINATE_STRING
Use of <code>memset</code> with size argument zero	MEMSET_INVALID_SIZE
Use of <code>new</code> or <code>make_unique</code> instead of more efficient <code>make_shared</code>	MISSING_MAKE_SHARED

Defect	Command-line Name
Use of non-secure temporary file	NON_SECURE_TEMP_FILE
Use of obsolete standard function	OBSOLETE_STD_FUNC
Use of path manipulation function without maximum sized buffer checking	PATH_BUFFER_OVERFLOW
Use of plain char type for numerical value	BAD_PLAIN_CHAR_USE
Use of previously closed resource	CLOSED_RESOURCE_USE
Use of previously freed pointer	FREED_PTR
Use of setjmp/longjmp	SETJMP_LONGJMP_USE
Use of signal to kill thread	THREAD_KILLED_WITH_SIGNAL
Use of tainted pointer	TAINTED_PTR
Use of tainted pointer	TAINTED_PTR
Use of undefined thread ID	UNDEFINED_THREAD_ID
Variable length array with nonpositive size	NON_POSITIVE_VLA_SIZE
Variable shadowing	VAR_SHADOWING
Vulnerable path manipulation	PATH_TRAVERSAL
Vulnerable permission assignments	DANGEROUS_PERMISSIONS
Vulnerable pseudo-random number generator	VULNERABLE_PRNG
Weak cipher algorithm	CRYPTO_CIPHER_WEAK_CIPHER
Weak cipher mode	CRYPTO_CIPHER_WEAK_MODE
Weak padding for RSA algorithm	CRYPTO_RSA_WEAK_PADDING
Write without a further read	USELESS_WRITE
Writing to const qualified object	CONSTANT_OBJECT_WRITE
Writing to read-only resource	READ_ONLY_RESOURCE_WRITE
Wrong allocated object size for cast	OBJECT_SIZE_MISMATCH
Wrong type used in sizeof	PTR_SIZEOF_MISMATCH
X.509 peer certificate not checked	CRYPTO_SSL_CERT_NOT_CHECKED

See Also

More About

- “Annotate Code and Hide Known or Acceptable Results” on page 30-2
- “Choose Specific Bug Finder Defect Checkers” on page 18-2

Short Names of Code Complexity Metrics

When annotating your code to justify metrics or creating custom software quality objectives, you use short names of code complexity metrics instead of the full names. The following table lists the short names for code complexity metrics.

Note that you can only annotate your code for function level code complexity metrics only.

Project Metrics

Code Metric	Acronym
Number of Direct Recursions	AP_CG_DIRECT_CYCLE
Number of Header Files	INCLUDES
Number of Files	FILES
Number of Recursions	AP_CG_CYCLE
Program Maximum Stack Usage	PROG_MAX_STACK
Program Minimum Stack Usage	PROG_MIN_STACK

File Metrics

Code Metric	Acronym
Comment Density	COMF
Estimated Function Coupling	FCO
Number of Lines	TOTAL_LINES
Number of Lines Without Comment	LINES_WITHOUT_CMT

Function Metrics

Code Metric	Acronym
Cyclomatic Complexity	VG
Higher Estimate of Size of Local Variables	LOCAL_VARS_MAX
Language Scope	VOCF
Lower Estimate of Size of Local Variables	LOCAL_VARS_MIN
Minimum Stack Usage	MIN_STACK
Maximum Stack Usage	MAX_STACK
Number of Call Levels	LEVEL
Number of Call Occurrences	NCALLS
Number of Called Functions	CALLS
Number of Calling Functions	CALLING

Code Metric	Acronym
Number of Executable Lines	FXLN
Number of Function Parameters	PARAM
Number of Goto Statements	GOTO
Number of Instructions	STMT
Number of Lines Within Body	FLIN
Number of Local Non-Static Variables	LOCAL_VARS
Number of Local Static Variables	LOCAL_STATIC_VARS
Number of Paths	PATH
Number of Return Statements	RETURN

See Also

More About

- “Annotate Code and Hide Known or Acceptable Results” on page 30-2

Annotate Code for Known or Acceptable Results (Not Recommended)

Note Starting R2017b, Polyspace uses a simpler annotation format. See “Annotate Code and Hide Known or Acceptable Results” on page 30-2.

If Polyspace finds defects in your code that you cannot or will not fix, you can add annotations to your code. These annotations are code comments that indicate known or acceptable defects or coding rule violations. By using these annotations, you can:

- Avoid rereviewing defects or coding rule violations from previous analyses.
- Preserve review comments and classifications.

Note Source code annotations do not apply to code comments. You cannot annotate these rules:

- MISRA C:2004 Rules 2.2 and 2.3
 - MISRA C:2012 Rules 3.1 and 3.2
 - MISRA-C++ Rule 2-7-1
 - JSF++ Rules 127 and 133
-

Add Annotations from the Polyspace Interface

This method shows you how to convert review comments and classifications in the Polyspace interface into code annotations.

- 1 On the **Results List** or **Result Details** pane, assign a **Severity**, **Status**, and **Comment** to a result.
 - a Click a result.
 - b From the **Severity** and **Status** dropdown lists, select an option.
 - c In the **Comment** field, enter a comment or keyword that helps you easily recognize the result.
- 2 On the **Results List** pane, right-click the commented result and select **Add Pre-Justification to Clipboard**. The software copies the severity, status, and comment to the clipboard.
- 3 Right-click the result again and select **Open Editor**. The software opens the source file to the location of the defect.
- 4 Paste the contents of your clipboard on the line immediately before the line containing the defect or coding rule violation.

You can see your review comments as a code comment in the Polyspace annotation syntax, which Polyspace uses to repopulate review comments on your next analysis.

- 5 Save your source file and rerun the analysis.

On the **Results List** pane, the software populates the **Severity**, **Status**, and **Comment** columns for the defect or rule violation that you annotated. These fields are read only because they are

populated from your code annotation. If you use a specific keyword or status for your annotations, you can filter your results to hide or show your annotated results. For more information on filtering, see “Filter and Group Results in Polyspace Desktop User Interface” on page 23-2.

Add Annotations Manually

This method shows you how to enter comments directly into your source files by using the Polyspace code annotation syntax. The syntax is not case-sensitive and applies to the first uncommented line of C/C++ code following the annotation.

- 1 Open your source file in an editor and locate the line or section of code that you want to annotate.
- 2 Add one of the following annotations:
 - For a single line of code, add the following text directly before the line of code that you want to annotate.

```
/* polyspace<Type:Kind1[,Kind2] : [Severity] : [Status] > [Notes] */
```

- For a section of code, use the following syntax.

```
/* polyspace:begin<Type:Kind1[,Kind2] : [Severity] : [Status] > [Notes] */
```

```
... Code section ...
```

```
/* polyspace:end<Type:Kind1[,Kind2] : [Severity] : [Status] > */
```

If a macro expands to multiple lines, use the syntax for code sections even though the macro itself covers one line. The single-line syntax applies only to results that appear in the first line of the expanded macro.

- 3 Replace the words *Type*, *Kind1*, [*Kind2*], [*Severity*], [*Status*], and [*Additional text*] with allowed values, indicated in the following table. The text with square brackets [] is optional and you can delete it. See “Syntax Examples” on page 30-28.

Word	Allowed Values
<i>Type</i>	<p>The type of results:</p> <ul style="list-style-type: none"> • Defect (Polyspace Bug Finder) • RTE, for run-time checks (Polyspace Code Prover) • VARIABLE, for global variables (Polyspace Code Prover) • CODE-METRIC, for code complexity metrics. • MISRA-C, for MISRA C:2004 • MISRA-AC-AGC • MISRA-C3, for MISRA C:2012 • MISRA-CPP • JSF • Custom, for custom coding rule violations.

Word	Allowed Values
<i>Kind1, [Kind2], ...</i>	<p>For defects, run-time checks and code metrics, use the short names of checkers. See:</p> <ul style="list-style-type: none"> • “Short Names of Bug Finder Defect Groups and Defect Checkers” on page 30-11 • “Short Names of Code Prover Run-Time Checks” (Polyspace Code Prover) <p>For coding rule violations, specify the rule number or numbers.</p> <p>For global variables, the only allowed value is ALL.</p> <p>If you want the comment to apply to all possible defects or coding rules, specify ALL.</p>
<i>Severity</i>	<p>Text that indicates how critical you consider the defect. Enter one of the following:</p> <ul style="list-style-type: none"> • Unset • High • Medium • Low <p>This text populates the Severity column on the Results List pane.</p>
<i>Status</i>	<p>Text that indicates how you intend to correct the error in your code. Enter one of the following or any other text:</p> <ul style="list-style-type: none"> • Unreviewed • To investigate • To fix • Justified • No action planned • Not a defect • Other <p>This text populates the Status column on the Results List pane. The status is also used in Polyspace Access to determine whether a result is justified. To justify a result, use Justified, No action planned or Not a defect.</p>
<i>Notes</i>	<p>Additional comments, such as a keyword or an explanation for the status and severity.</p>

Syntax Examples

- A single defect:

```
/* polyspace<Defect:HARD_CODED_BUFFER_SIZE:Medium:To investigate> Known issue */
int table[100];
```

- A single run-time check:

```
/* polyspace<RTE: ZDV : High : To Fix > Denominator cannot be zero */
y=1/x;
```

- A MISRA C:2012 rule violation:

```
/* polyspace<MISRA-C3: 13.1 : Low : Justified> Known issue */  
int arr[2] = {x++,y};
```

- Unused global variable:

```
/* polyspace<VARIABLE: ALL : Low : Justified> Variable to use later*/  
int var_unused;
```

- Multiple defects:

```
polyspace<Defect:USELESS_WRITE,DEAD_CODE:Low:No Action Planned> OK issue
```

- Multiple JSF rule violations:

```
polyspace<JSF:9,13:Low:Justified> Known issue
```

Define Custom Annotation Format

This example shows how to create and edit an XML file to define an annotation format and map it to the Polyspace annotation syntax. Once you create and edit the XML file, pass the file to Polyspace by using option `-xml-annotations-description`.

To define multiple custom annotation formats, see “Define Multiple Custom Annotation Syntaxes” on page 30-36.

To get started, copy the following code to a text editor and save it on your machine as `annotations_description.xml`.


```

<?xml version="1.0" encoding="UTF-8"?>
<Annotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="annotations_xml_schema.xsd"
  Group="example XML">

  <Expressions Search_For_Keywords="myKeyword"
    Separator_Result_Name="," >
    <!-- Define annotation format in this
    section by adding <Expression/> elements -->

    <Expression Mode="SAME_LINE"
      Regex="myKeyword\s+(\w+(\s*,\s*\w+)*)"
      Rule_Identifier_Position="1"
    />

    <Expression Mode="GOTO_INCREMENT"
      Regex="myKeyword\s+(\d+\s)(\w+(\s*,\s*\w+)*)"
      Increment_Position="1"
      Rule_Identifier_Position="2"
    />

    <Expression Mode="BEGIN"
      Regex="myKeyword\s*(\w+(\s*,\s*\w+)*)\s*Block_on"
      Rule_Identifier_Position="1"
      Case_Insensitive="true"
    />

    <Expression Mode="END"
      Regex="myKeyword\s*(\w+(\s*,\s*\w+)*)\s*Block_off"
      Rule_Identifier_Position="1"
    />

    <Expression Mode="END_ALL"
      Regex="myKeyword\sBlock_off_all"
    />

    <Expression Mode="SAME_LINE"

    Regex="myKeywords\s+(\w+(\s*,\s*\w+)*)
    (\s*\[(\w+\s*)*([:\s*(\w+\s*)+)*\])*(\s*-\s*)*(\[^\]*)(\s*-)*"
    Rule_Identifier_Position="1"
    Status_Position="4"
    Severity_Position="6"
    Comment_Position="8"
    />
    <!-- Put the regular expression on a single line instead of two line
    when you copy it to a text editor -->

    <!-- SAME_LINE example with more complex regular expression.
    Matches the following annotations:
    //myKeywords 50 [my_status:my_severity] -Additional comment-
    //myKeywords 50 [my_status]
    //myKeywords 50 [:my_severity]
    //myKeywords 50 -Additional comment-
    -->

  </Expressions>

  <Mapping>
    <!-- Map your annotation syntax to the Polyspace annotation
    syntax by adding <Result_Name_Mapping /> elements in this section -->

    <Result_Name_Mapping Rule_Identifier="100" Family="DEFECT"
      Result_Name="INT_ZERO_DIV"/>

    <Result_Name_Mapping Rule_Identifier="50" Family="MISRA-C3" Result_Name="8.4"/>
    <Result_Name_Mapping Rule_Identifier="51" Family="MISRA-C3" Result_Name="8.7"/>
    <Result_Name_Mapping Rule_Identifier="ALL_MISRA" Family="MISRA-C3" Result_Name="*"/>
  </Mapping>
</Annotations>

```

The XML file consists of two parts:

- <Expressions>...</Expressions> where you define the format of your annotation syntax.

- `<Mapping>...</Mapping>` where you map your syntax to the Polyspace annotation syntax.

After you edit this file, Polyspace can interpret your custom code annotation when you invoke the option `-xml-annotations-description`.

Define Annotation Syntax Format

To define an annotation syntax in Polyspace, your syntax must follow a pattern that you can represent with a regular expression. See “Regular Expressions”. It is recommended that you include a keyword in the pattern of your annotation syntax to help identify it. In this example, the keyword is `myKeyword`. Set the attribute `Search_For_Keywords` equal to this keyword.

Once you know the pattern of your annotation, you can define it in the XML by adding an `<Expression/>` element and specifying at least the attributes `Mode`, `Regex`, and `Rule_Identifier_Position`. For instance, the first `<Expression/>` element in `annotations_description.xml` defines an annotation with these attributes:

- `Mode="SAME_LINE"`. The annotation applies to code on the same line.
- `Regex="myKeyword\s+(\w+(\s*,\s*\w+)*)"`. Polyspace uses the regular expression to search for a string that begins with `myKeyword`, followed by a space `\s+`. Polyspace then searches for a capturing group `(\w+(\s*,\s*\w+)*)` that includes an alphanumeric rule identifier `\w+` and, optionally, additional comma-separated rule identifiers `(\s*,\s*\w+)*`.
- `Rule_Identifier_Position="1"`. The integer value of this attribute corresponds to the number of opening parentheses preceding the relevant capturing group in the regular expression. In `myKeyword\s+(\w+(\s*,\s*\w+)*)`, one opening parenthesis precedes the capturing group of the rule identifier `(\w+(\s*,\s*\w+)*)`. If you want to match rule identifiers captured by `(\s*,\s*\w+)`, then you set `Rule_Identifier_Position="2"` because two opening parentheses precede this capturing group.

The list of attributes and their values are listed in this table. The example column refers to the format defined in `annotations_description.xml`.

Attribute	Use	Value	Example
Mode	Required	SAME_LINE	Applies only on the same line as the annotation. code; //myKeyword 100
		GOTO_INCREMENT	Applies on the same line as the annotation and the following n lines: 3. code; // myKeyword +3 ALL_MISRA 4. /*comments */ 5. 6. code; 7. code; The preceding annotation applies to lines 3-6 only.

Attribute	Use	Value	Example
		BEGIN	<p>Applies to the same line and all following lines until a corresponding expression with attribute Mode="END" or "END_ALL", or until the end of the file.</p> <pre>//myKeyword 50, 51 Block_on Code block 1; ...</pre>
		END	<p>Stops the application of a rule identifier declared by a corresponding expression with attribute Mode="BEGIN".</p> <pre>//myKeyword 50, 51 Block_on Code block 1; ... More code; //myKeyword 50 Block_off</pre> <p>Only rule identifier 50 is turned off. Rule identifier 51 still applies.</p>
		END_ALL	<p>Stops all rule identifiers declared by an expression with attribute Mode="BEGIN".</p> <pre>//myKeyword 50, 51 Block_on Code block 1; ... More code; //myKeyword Block_off_all</pre> <p>Rule identifiers 50 and 51 are turned off.</p>
Regex	Required	Regular expression search string	<p>See "Regular Expressions". Regex="myKeyword\s+(\w+(\s*,\s*\w+)*)" matches these expressions:</p> <pre>// myKeyword 50, 51 /* myKeyword ALL_MISRA, 100 */</pre>

Attribute	Use	Value	Example
Rule_Identifier_Position	Required, except when you set Mode="END_ALL"	Integer	<p>The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.</p> <pre><Expression Mode="GOTO_INCREMENT" Regex="myKeyword\s+(\+\d+\s) (\w+(\s*,\s*\w+))*" Increment_Position="1" Rule_Identifier_Position="2"/></pre> <p>Note Enter the regex expression on a single line when you edit your XML file.</p> <p>The search expression for the rule identifier <code>\w+(\s*,\s*\w+)*</code> is after the second opening parenthesis of the regular expression.</p>
Increment_Position	Required only when you set Mode="GOTO_INCREMENT"	Integer	<p>The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.</p> <pre><Expression Mode="GOTO_INCREMENT" Regex="myKeyword\s+(\+\d+\s) (\w+(\s*,\s*\w+))*" Increment_Position="1" Rule_Identifier_Position="2"/></pre> <p>Note Enter the regex expression on a single line when you edit your XML file.</p> <p>The search expression for the increment <code>\+\d+\s</code> is after the first opening parenthesis of the regular expression.</p>
Status_Position	Optional	Integer	<p>See Increment_Position example. When you use this attribute, the entry in your annotation is displayed in the Status column on the Results List pane of the user interface.</p>
Severity_Position	Optional	Integer	<p>See Increment_Position example. When you use this attribute, the entry in your annotation is displayed in the Severity column on the Results List pane of the user interface.</p>

Attribute	Use	Value	Example
Comment_Position	Optional	Integer	See Increment_Position example. When you use this attribute, the entry in your annotation is displayed in the Comment column on the Results List pane of the user interface. Your comment is appended to the string <code>Justified by annotation in source:</code>
Case_Insensitive	Optional	True or false	When you set this attribute to "true", the regular expression is case insensitive, otherwise it is case sensitive. If you do not declare this attribute in your expression, the regular expression is case sensitive. For <code>Case_Insensitive="true"</code> , these annotations are equivalent: <pre>//MYKEYWORD ALL_MISRA BLOCK_ON //mykeyword all_misra block_on</pre>

Map Your Annotation to the Polyspace Annotation Syntax

After you define your annotation format, you can map the rule identifiers you are using to their corresponding Polyspace annotation syntax. You can do this mapping by adding an `<Result_Name_Mapping/>` element and specifying attributes `Rule_Identifier`, `Family`, and `Result_Name`. For instance, if rule identifier 50 corresponds to MISRA C: 2012 rule 8.4, map it to the Polyspace syntax MISRA-C3:8.4 by using this element:

```
<Result_Name_Mapping Rule_Identifier="50" Family="MISRA-C3" Result_Name="8.4"/>
```

The list of attributes and their values are listed in this table. The example column refers to the format defined in `annotations_description.xml`.

Attribute	Use	Value	Example
Rule_Identifier	Required	User defined. Each value must be unique.	See the mapping section of <code>annotations_description.xml</code>
Family	Required	Corresponds to Polyspace results family. For a list of allowed values, see allowed values on page 30-2.	See the mapping section of <code>annotations_description.xml</code>
Result_Name	Required	Corresponds to Polyspace result names. For a list of allowed values, see allowed values on page 30-2.	See the mapping section of <code>annotations_description.xml</code>

Define Multiple Custom Annotation Syntaxes

To define more than one annotation syntax, in your XML file, specify a comma separated list of keywords associated with each syntax for the `Search_For_Keywords` attribute.

For example, if you use custom annotations that follow these patterns to annotate violations of MISRA C: 2012 rules:

```
int func(int p) //customSyntax M123 $ customSyntax M124
{
    int i;
    int j = 1;

    i = 1024 / (j - p);
    return i;
}

int func2(void){ //otherCustomSyntax 50
    int x=func(2);
    return x;
}
```

Enter the following in the XML file where you define the custom annotation syntax.

```
<?xml version="1.0" encoding="UTF-8"?>
<Annotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="annotations_xml_schema.xsd"
  Group="multipleCustomSyntax">
  <!-- Enter comma separated list of keywords -->
  <Expressions Search_For_Keywords="customSyntax,otherCustomSyntax"
    Separator_Result_Name="$" >

    <!-- This section defines the annotation syntax format -->
    <Expression Mode="SAME_LINE"
      Regex="customSyntax\s(\w+(\s*,\s*\w+)*)"
      Rule_Identifier_Position="1"
    />
    <Expression Mode="SAME_LINE"
      Regex="otherCustomSyntax\s(\w+(\s*,\s*\w+)*)"
      Rule_Identifier_Position="1"
    />
  </Expressions>
  <!-- This section maps the user annotation to the Polyspace
  annotation syntax -->
  <Mapping>
    <!-- Mapping for customSyntax rules -->
    <Result_Name_Mapping Rule_Identifier="M123" Family="MISRA-C3" Result_Name="8.7"/>
    <Result_Name_Mapping Rule_Identifier="M124" Family="MISRA-C3" Result_Name="D4.6"/>
    <!-- Mapping for otherCustomSyntax rules -->
    <Result_Name_Mapping Rule_Identifier="50" Family="MISRA-C3" Result_Name="8.4"/>
  </Mapping>
</Annotations>
```

When you use multiple custom annotations, each rule identifier must be unique. For instance, in the preceding example, you cannot reuse rule identifier M123 with otherCustomSyntax.

See Also

-xml-annotations-description

More About

- “Annotation Description Full XML Template” on page 30-38
- “Annotate Code and Hide Known or Acceptable Results” on page 30-2

- “Fix Errors Applying Custom Annotation Format for Polyspace Results” on page 32-63

Annotation Description Full XML Template

This table lists all the elements, attributes, and values of the XML that you can use to define an annotation format and map it to the Polyspace annotation syntax. For an example of how to edit an XML to define annotation syntax, see “Define Custom Annotation Format” on page 30-30.

Element	Attribute	Use	Value
Annotations	Group	Required	User defined string. For example, "Custom Annotations"
Expressions	Search_For_Keyword s	Required	User defined string. This string is a keyword you include in the pattern of your annotation syntax to help identify it. For example, "myKeyword". To use multiple custom annotations, enter a comma separated list of keyword. See “Define Multiple Custom Annotation Syntaxes” on page 30-36.
	Separator_Result_Name	Required	User defined string. This string is a separator when you list multiple Polyspace result names in the same annotation. For example ","
	Separator_Family_And_Result_Name	Optional	User defined string. This string is a separator when you list multiple Polyspace results families in the same annotation. For example, " "
	Separator_Family	Optional	User defined string. This string is a separator when you list a Polyspace results family and results name in the same annotation. For example, ":"
Expression	Mode	Required	SAME_LINE
			GOTO_INCREMENT
			BEGIN

Element	Attribute	Use	Value
			END
			END_ALL
			NEXT_CODE_LINE
			The annotation applies to the next line of code. Comments and blank lines are ignored.
			GOTO_LABEL
			LABEL
			XML_START
			XML_CONTENT
			The annotation for this expression must be on a single line.
	XML_END		
	Regex	Required	Regular expression search string that matches the pattern of your annotation.
	Rule_Identifier_Position	Required, except when you set Mode="END_ALL" or "LABEL"	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Increment_Position	Required only when you set Mode="GOTO_INCREMENT"	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Status_Position	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.

Element	Attribute	Use	Value
	Severity_Position	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Comment_Position	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Label_Position	Required only when you set Mode="GOTO_LABEL" or "LABEL"	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Case_Insensitive	Optional	True or false. When you do not declare this attribute, the default value is false.
	Is_Pragma	Optional	True or false. When you do not declare this attribute, the default value is false. Set this attribute to true if you want to declare your annotation using a pragma instead of a comment.
	Applies_Also_On_Same_Line	Optional	True or false. When you do not declare this attribute, the default value is true. Use this attribute to enable annotations with the old Polyspace syntax to apply on the same line of code.

Element	Attribute	Use	Value
Mapping	None	None	None
Result_Name_Mapping	Rule_Identifier	Required	User defined
	Family	Required	Corresponds to Polyspace results family. For a list of allowed values, see allowed values on page 30-2.
	Result_Name	Required	Corresponds to Polyspace result names. For a list of allowed values, see allowed values on page 30-2.

Example

This example code covers some of the less commonly used attributes for defining annotations in XML.

```

<?xml version="1.0" encoding="UTF-8"?>

<Annotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="annotations_xml_schema.xsd"
  Group="XML Template">

  <Expressions Separator_Result_Name="."
    Search_For_Keywords="myKeyword">

    <Expression Mode="GOTO_LABEL"
      Regex="(\\A|\\W)myKeyword\\s+S\\s+(\\d+(\\s*,\\s*\\d+)*\\s+([a-zA-Z_-]\\w+)"
      Rule_Identifier_Position="2"
      Label_Position="4"

      />

    <Expression Mode="LABEL"
      Regex="(\\A|\\W)myKeyword\\s+L:(\\w+)"
      Label_Position="2"

      />

    <!-- Annotation applies starting current line until
      next declaration of label word "myLabel"
      Example:

      code; // myKeyword S 100 myLabel
      ...
      more code;
      // myKeyword L myLabel
    -->

    <Expression Mode="BEGIN"
      Regex="#\\s*pragma\\s+myKeyword_MESSAGES_ON\\s+(\\w+)"
      Rule_Identifier_Position="1"
      Is_Pragma="true"
      />

    <!-- Annotation declared with pragma instead of comment
      Example:#pragma myKeyword_MESSAGES_ON 100 -->

    <!-- Comment declaration with XML format-->

    <!-- XML_START must be declared before XML_CONTENT -->
    <Expression Mode="XML_START"
      Regex="<\\s*myKeyword_COMMENT\\s*>"

      />

    <!-- Example: <myKeyword_COMMENT> -->

    <Expression Mode="XML_CONTENT"
      Regex="<\\s*(\\d*)\\s*>(((?![*/])(?!<).)*</\\s*(\\d*)\\s*>"
      Rule_Identifier_Position="1"
      Comment_Position="2"

      />

    <!-- Example: <100>This is my comment</100>
      XML_CONTENT must be declare on a single line.

      <100>This is my comment
      </100>
      is incorrect.
    -->

    <Expression Mode="XML_END"
      Regex="</\\s*myKeyword_COMMENT\\s*>"

      />

    <!-- Example: </myKeyword_COMMENT> -->
  </Expressions>

  <Mapping>

  <Result_Name_Mapping Rule_Identifier="100" Family="MISRA-C" Result_Name="4.1"/>
  </Mapping>
</Annotations>

```

See Also

-xml-annotations-description

More About

- “Annotate Code and Hide Known or Acceptable Results” on page 30-2

Advanced Review Workflows

Evaluate Polyspace Bug Finder Results Against Bug Finder Quality Objectives

Instead of evaluating all results of a Bug Finder analysis, you can first define a set of criteria that the analyzed project must meet and compare the Bug Finder results against those criteria. The Bug Finder Quality Objectives or BF-QOs are a set of thresholds against which you can compare your Bug Finder analysis results. You can develop a review process based on the Bug Finder Quality Objectives. In your review process, you consider only those results that cause your project to fail a certain BF-QO level.

The Bug Finder Quality Objectives are adapted from Software Quality Objectives in Code Prover. See “Evaluate Polyspace Code Prover Results Against Software Quality Objectives” (Polyspace Code Prover).

You can use a predefined BF-QO level or define your own. To customize BF-QO levels, see “Customize Software Quality Objectives” on page 25-16.

Following are the predefined quality thresholds specified by each BF-QO.

BF-QO Level 1

Metric	Threshold Value
Comment density of a file	20
Number of paths through a function	80
Number of <code>goto</code> statements	0
Cyclomatic complexity	10
Number of calling functions	5
Number of calls	7
Number of parameters per function	5
Number of instructions per function	50
Number of call levels in a function	4
Number of <code>return</code> statements in a function	1
Language scope, an indicator of the cost of maintaining or changing functions. Calculated as follows: $(N1+N2) / (n1+n2)$	4
<ul style="list-style-type: none"> • <i>n1</i> — Number of different operators • <i>N1</i> — Total number of operators • <i>n2</i> — Number of different operands • <i>N2</i> — Total number of operands 	
Number of recursions	0
Number of direct recursions	0

Metric	Threshold Value
Number of unjustified violations of the following MISRA C:2004 rules: <ul style="list-style-type: none"> • 5.2 • 8.11, 8.12 • 11.2, 11.3 • 12.12 • 13.3, 13.4, 13.5 • 14.4, 14.7 • 16.1, 16.2, 16.7 • 17.3, 17.4, 17.5, 17.6 • 18.4 • 20.4 	0
Number of unjustified violations of the following MISRA C:2012 rules: <ul style="list-style-type: none"> • 8.8, 8.11, and 8.13 • 11.1, 11.2, 11.4, 11.5, 11.6, and 11.7 • 14.1 and 14.2 • 15.1, 15.2, 15.3, and 15.5 • 17.1 and 17.2 • 18.3, 18.4, 18.5, and 18.6 • 19.2 • 21.3 	0
Number of unjustified violations of the following MISRA C++ rules: <ul style="list-style-type: none"> • 2-10-2 • 3-1-3, 3-3-2, 3-9-3 • 5-0-15, 5-0-18, 5-0-19, 5-2-8, 5-2-9 • 6-2-2, 6-5-1, 6-5-2, 6-5-3, 6-5-4, 6-6-1, 6-6-2, 6-6-4, 6-6-5 • 7-5-1, 7-5-2, 7-5-4 • 8-4-1 • 9-5-1 • 10-1-2, 10-1-3, 10-3-1, 10-3-2, 10-3-3 • 15-0-3, 15-1-3, 15-3-3, 15-3-5, 15-3-6, 15-3-7, 15-4-1, 15-5-1, 15-5-2 • 18-4-1 	0

BF-QO Level 2 and 3

In addition to all the requirements of BF-QO Level 1, these levels includes the following thresholds:

Metric	Threshold Value
Number of high-impact defects. See also "Classification of Defects by Impact" on page 18-49.	0

BF-QO Level 4

In addition to all the requirements of BF-QO Level 2 and 3, this level includes the following thresholds:

Metric	Threshold Value
Number of medium-impact defects. See also "Classification of Defects by Impact" on page 18-49.	0

BF-QO Level 5

In addition to all the requirements of BF-QO Level 4, this level includes the following thresholds:

Metric	Threshold Value
Number of unjustified violations of the following MISRA C:2004 rules: <ul style="list-style-type: none"> • 6.3 • 8.7 • 9.2, 9.3 • 10.3, 10.5 • 11.1, 11.5 • 12.1, 12.2, 12.5, 12.6, 12.9, 12.10 • 13.1, 13.2, 13.6 • 14.8, 14.10 • 15.3 • 16.3, 16.8, 16.9 • 19.4, 19.9, 19.10, 19.11, 19.12 • 20.3 	0
Number of unjustified violations of the following MISRA C:2012 rules: <ul style="list-style-type: none"> • 11.8 • 12.1 and 12.3 • 13.2 and 13.4 • 14.4 • 15.6 and 15.7 • 16.4 and 16.5 • 17.4 • 20.4, 20.6, 20.7, 20.9, and 20.11 	0

Metric	Threshold Value
Number of unjustified violations of the following MISRA C++ rules: <ul style="list-style-type: none"> • 3-4-1, 3-9-2 • 4-5-1 • 5-0-1, 5-0-2, 5-0-7, 5-0-8, 5-0-9, 5-0-10, 5-0-13, 5-2-1, 5-2-2, 5-2-7, 5-2-11, 5-3-3, 5-2-5, 5-2-6, 5-3-2, 5-18-1 • 6-2-1, 6-3-1, 6-4-2, 6-4-6, 6-5-3 • 8-4-3, 8-4-4, 8-5-2, 8-5-3 • 11-0-1 • 12-1-1, 12-8-2 • 16-0-5, 16-0-6, 16-0-7, 16-2-2, 16-3-1 	0

BF-QO Level 6

In addition to all the requirements of BF-QO Level 5, this level includes the following thresholds:

Metric	Threshold Value
Number of low-impact defects. See also “Classification of Defects by Impact” on page 18-49.	0

BF-QO Exhaustive

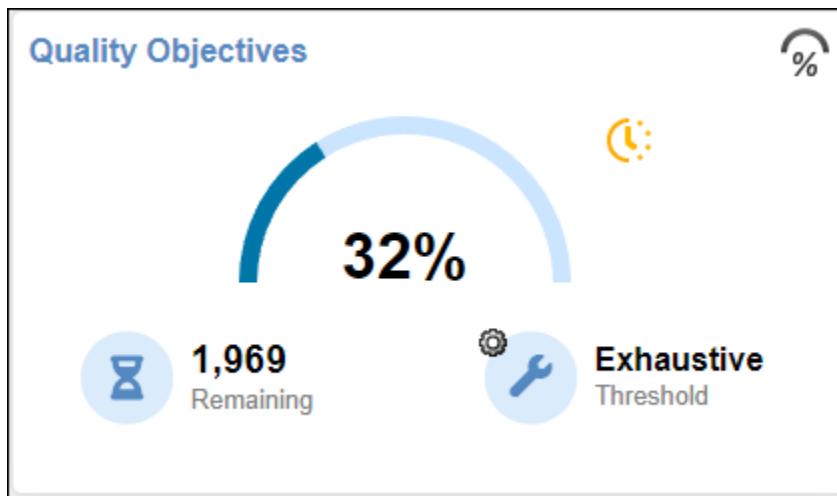
In addition to all the requirements of BF-QO Level 6, this level includes the following thresholds. The thresholds for coding rule violations apply only if you check for coding rule violations.

Metric	Threshold Value
Number of unjustified MISRA C and MISRA C++ coding rule violations	0
Number of unjustified defects	0

Comparing Analysis Results Against Quality Objectives

You can compare your analysis results against SQOs either in the Polyspace Access web interface or the Polyspace user interface.

- In the Polyspace Access web interface, you can first determine whether your project fails to attain a certain Quality Objective threshold by looking at the **Quality Objectives** card on the **Project Overview** dashboard.



The card shows the percentage of results that you have already fixed or justified in order to attain the threshold. Click the number of remaining findings to open those findings in the **Results List**. For a more detailed view of the quality of your code against all quality objectives thresholds, open the **Quality Objectives** dashboard. For more information, see the “Quality Objectives Dashboard in Polyspace Access” on page 25-14.

- In the Polyspace user interface, you can use the menu in the **Results List** toolbar to display only those results that you must fix or justify to attain a certain Software Quality Objective.

To activate the SQO options in this menu, select **Tools > Preferences**. On the **Review Scope** tab, select **Include Quality Objectives Scope**.

See Also

Related Examples

- “Filter and Sort Results in Polyspace Access Web Interface” on page 27-8
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 26-2

Justify Coding Rule Violations Using Code Prover Checks

Coding rules are good practices that you observe for safe and secure code. Using the Polyspace coding rule checkers, you find instances in your code that violate a coding rule standard such as MISRA. If you run Code Prover, you also see results of checks that find run-time errors or prove their absence. In some cases, the two kinds of results can be used together for efficient review. For instance, you can use a green Code Prover check as rationale for not fixing a coding rule violation (justification).

If you run MISRA checking in Code Prover, some of the checkers use Code Prover static analysis under the hood to find MISRA violations. The MISRA checker in Code Prover is more rigorous compared to Bug Finder because Code Prover keeps precise track of the data and control flow in your code. For instance:

- MISRA C:2012 Rule 9.1 (Polyspace Code Prover): The rule states that the value of an object with automatic storage duration shall not be read before it has been set. Code Prover uses the results of a `Non-initialized local variable` (Polyspace Code Prover) check to determine the rule violations.
- MISRA C:2004 Rule 13.7: The rule states that the Boolean operations whose results are invariant shall not be permitted. Code Prover uses the results of an `Unreachable code` (Polyspace Code Prover) check to identify conditions that are always true or false.

In some other cases, the MISRA checkers do not suppress rule violations even though corresponding green checks indicate that the violations have no consequence. You have the choice to do one of these:

- Strictly conform to the standard and fix the rule violations.
- Manually justify the rule violations using the green checks as rationale.

Set a status such as `No action planned` to the result and enter the green check as rationale in the result comments. You can later filter justified results using that status.

The following sections show examples of situations where you can justify MISRA violations using green Code Prover checks.

Rules About Data Type Conversions

In some cases, implicit data type conversions are okay if the conversion does not cause an overflow.

In the following example, the line `temp = var1 - var2;` violates MISRA C:2012 Rule 10.3 (Polyspace Code Prover). The rule states that the value of an expression shall not be assigned to an object of a different essential type category. Here, the difference between two `int` variables is assigned to a `char` variable. You can justify this particular rule violation by using the results of a Code Prover `Overflow` (Polyspace Code Prover) check.

```

int func (int var1, int var2) {
    char temp;
    temp = var1 - var2;
    if (temp > 0)
        return -1;
    else
        return 1;
}

double read_meter1(void);
double read_meter2(void);

int main(char arg, char* argv[]) {
    int meter1 = (read_meter1()) * 10;
    int meter2 = (read_meter2()) * 999;
    int tol = 10;
    if((meter1 - meter2)> -tol && (meter1 - meter2) < tol)
        func(meter1, meter2);
    return 0;
}

```

Consider the rationale behind this rule. The use of implicit conversions between types can lead to unintended results, including possible loss of value, sign, or precision. For a conversion from `int` to `char`, a loss of sign or precision cannot happen. The only issue is a potential loss of value if the difference between the two `int` variables overflows.

Run Code Prover on this code. On the **Source** pane, click the `=` in `temp = var1 - var2;`. You see the expected violation of MISRA C:2012 Rule 10.3, but also a green **Overflow** check.

Select one or more results to review:

- ✓ **Overflow**
- ▼ MISRA C:2012 10.3 (Required)
- ▼ MISRA C:2012 10.3 (Required) ?

The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category. The expression is assigned to an object with a different essential type category.

Source

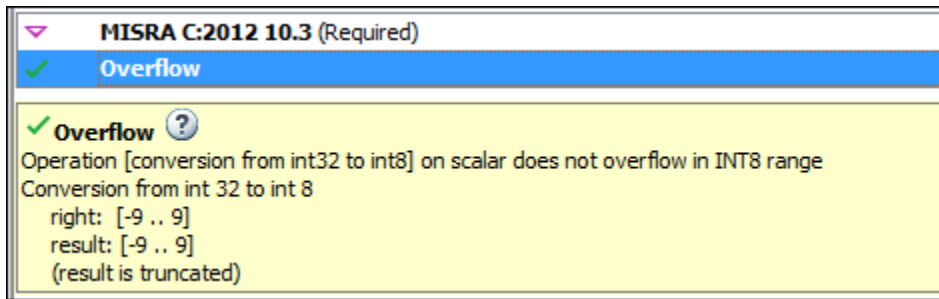
file2.c x

```

1  int func (int var1, int var2) {
2      char temp;
3      temp = var1 - var2;
4      if (temp > 0)
5          return -1;
6      else
7          return 1;
8  }

```

The green check indicates that the conversion from `int` to `char` does not overflow.



You can use the green overflow check as rationale to justify the rule violation.

Rules About Pointer Arithmetic

Pointer arithmetic on nonarray pointers are okay if the pointers stay within the allowed buffer.

In the following example, the operation `ptr++` violates MISRA C:2004 Rule 17.4. The rule states that array indexing shall be the only allowed form of pointer arithmetic. Here, a pointer that is not an array is incremented.

```

#define NUM_RECORDS 3
#define NUM_CHARACTERS 6

void readchar(char);

int main(int argc, char* argv[]) {
    char dbase[NUM_RECORDS][NUM_CHARACTERS] = { "r5cvx", "a2x5c", "g4x3c" };
    char *ptr = &dbase[0][0];
    for (int index = 0; index < NUM_RECORDS * NUM_CHARACTERS; index++) {
        readchar(*ptr);
        ptr++;
    }
    return 0;
}

```

Consider the rationale behind this rule. After an increment, a pointer can go outside the bounds of an allowed buffer (such as an array) or even point to an arbitrary location. Pointer arithmetic is fine as long as the pointer points within an allowed buffer. You can justify this particular rule violation by using the results of a Code Prover `Illegally dereferenced pointer` (Polyspace Code Prover) check.

Run Code Prover on this code. On the **Source** pane, click the `++` in `ptr++`. You see the expected violation of MISRA C:2004 Rule 17.4.

Result Review

Severity Enter comment here...
Status

MISRA C:2004 17.4 (Required) ?
Array indexing shall be the only allowed form of pointer arithmetic.

Configuration Result Details

Source

file3.c X

```

1  #define NUM_RECORDS 3
2  #define NUM_CHARACTERS 6
3
4  void readchar(char);
5
6  int main(int argc, char* argv[]) {
7  M   char dbase[NUM_RECORDS][NUM_CHARACTERS] = { "r5cvx", "a2x5c", "g4x3c"};
8     char *ptr = &dbase[0][0];
9  M   for (int index = 0; index < NUM_RECORDS * NUM_CHARACTERS; index++) {
10      readchar(*ptr);
11      ptr++;
12  }
13  return 0;
14  }
```

Click the * on the operation `readchar(*ptr)`. You see a green **Illegally dereferenced pointer** check. The green check indicates that the pointer points within allowed bounds when dereferenced.

✓ **Illegally dereferenced pointer** ?

Pointer is within its bounds

Dereference of local pointer 'ptr' (pointer to int 8, size: 8 bits):

- Pointer is not null.
- Points to 1 bytes at offset [0 .. 17] in buffer of 18 bytes, so is within bounds (if memory is allocated).
- Pointer may point to variable or field of variable:
 - 'dbase', local to function 'main'.

You can use the green check to justify the rule violation.

See Also

Related Examples

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 22-2

Polyspace Results in Lines Containing Macros


Macros in C/C++ can improve readability and maintainability of code. A macro is a named fragment of code defined with the `#define` directive, for instance:

```
#define MAXSIZE 64
```

The macro name acts as a shorthand for the fragment of code. During preprocessing, each instance of a macro is replaced with its definition. For instance, in the above example, each time you use `MAXSIZE`, it is replaced with `64` during preprocessing.

Polyspace provides several conveniences for reviewing results in lines containing macros.

Macros in Source Lines Can Be Expanded in Place

If a source code line contains a macro, the **Source** pane displays the line with an  icon on the left. You can click the icon to expand the macro, that is, see the macro definition, and click again to collapse the macro. See also:

- Bug Finder:
 - “Source Code in Polyspace Desktop User Interface” on page 21-17
 - “Source Code in Polyspace Access Web Interface” on page 25-21
- Code Prover:
 - “Source Code in Polyspace Desktop User Interface” (Polyspace Code Prover)
 - “Source Code in Polyspace Access Web Interface” (Polyspace Code Prover)

If a macro expansion contains multiple Code Prover run-time checks, the line with the macro collapsed has the same color as the worst run-time check. See also “Code Prover Result and Source Code Colors” (Polyspace Code Prover).

Results in Function-Like Macros Shown Only Once

A function-like macro is a macro that takes parameters, for instance:

```
#define max(x,y) x>y?x:y
```

If a function-like macro causes a defect or coding standard violation, the result is displayed on the root cause of the issue: the macro parameter or the macro definition.

For instance:

- In this example, the definition of macro `LEFTOVER()` contains a lowercase `l` and violates MISRA C:2012 Rule 7.3. This result is shown on the macro definition.

```
#define LEFTOVER(size) 10000ul - size /* Noncompliant */
#define REMAINDER(size) 10000UL - size /* Compliant */

void func(int arrSize, int arrCopySize) {
    int n = LEFTOVER(arrSize);
    int nCopy = LEFTOVER(arrCopySize);
    int m = REMAINDER(arrSize);
}
```

The event list below the result message shows the instances where the macro is used. You can click on an **Expansion of macro** event to navigate to the macro usage in the source code.

Event	File	Scope	Line
1 Expansion of macro	file.c	func()	5
2 Expansion of macro	file.c	func()	6
3 ▼ MISRA C:2012 7.3	file.c	File Scope	1

- In this example, the definition of macro `COPY_ELEMENT()` results in an ambiguous evaluation order and violates MISRA C:2012 Rule 13.2 only when the parameter `i++` is passed to it. This result is shown on the macro expansion, specifically on the parameter in the expansion.

```
int a[10], b[10];
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])

void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);          /* Compliant */
    COPY_ELEMENT (i++);       /* Noncompliant */
}
```

This way of showing results in function-like macros enables you to easily fix them:

- For issues caused by the macro definition, you can implement the fix once. Tools that report on the macro expansion can show multiple violations for one root cause.

In the preceding example, you can change the lowercase `l` in `LEFTOVER()` to fix the issue. The `REMAINDER()` macro shows this fix.

- For issues caused by the macro parameters, you can also implement the fix once.

In the preceding example, you can compute `i++` in a separate step, and then pass `i` to the `COPY_ELEMENT()` macro to fix the issue.

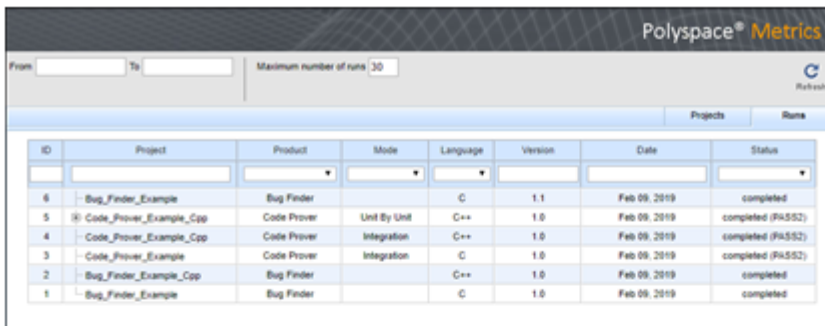
Migrate Results from Polyspace Metrics to Polyspace Access

If you use Polyspace Metrics to store results and monitor the quality of your source code, you can transfer those results to Polyspace Access.

The Polyspace Access **DASHBOARD** perspective offers a web interface with navigation between projects and categories of results. From the **Project Overview** dashboard, view aggregated statistics for all your projects or drill down to view results details by category or file. For each family of findings, open an additional dashboard to see details. After you narrow down the set of findings that you want to address, open them in the **REVIEW** perspective to start reviewing individual results.

Note The **REVIEW** perspective is only available for analysis results generated with a Polyspace product version R2019a or later. To review R2018b or earlier results that you migrated to Polyspace Access, see “Open or Export Results from Polyspace Access” on page 28-2.

You can also review results from Polyspace Access by opening them in the Polyspace desktop interface. You do not need to download a local copy of Polyspace Access results to view those results in the desktop interface. The edits that you make to the results are saved directly in Polyspace Access and enable you to perform collaborative reviews.



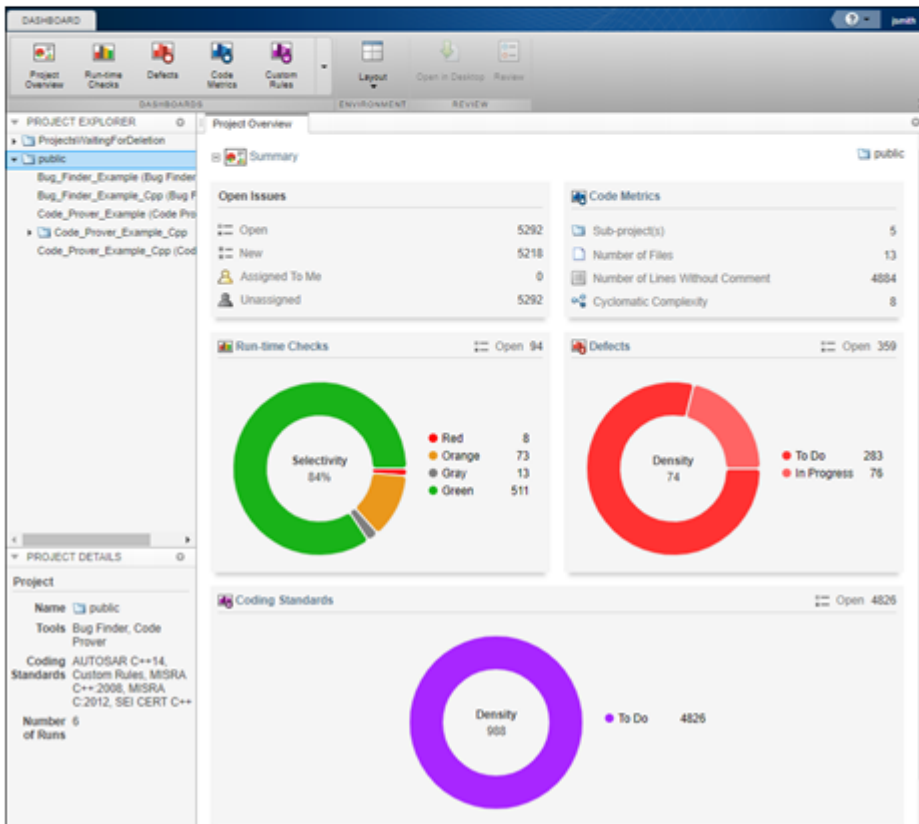
Polyspace Metrics

From To Maximum number of runs 30 Refresh

ID	Project	Product	Mode	Language	Version	Date	Status
6	-Bug_Finder_Example	Bug Finder		C	1.1	Feb 09, 2019	completed
5	Code_Prover_Example_Cpp	Code Prover	Unit By Unit	C++	1.0	Feb 09, 2019	completed (P4552)
4	-Code_Prover_Example_Cpp	Code Prover	Integration	C++	1.0	Feb 09, 2019	completed (P4552)
3	-Code_Prover_Example	Code Prover	Integration	C	1.0	Feb 09, 2019	completed (P4552)
2	-Bug_Finder_Example_Cpp	Bug Finder		C++	1.0	Feb 09, 2019	completed
1	-Bug_Finder_Example	Bug Finder		C	1.0	Feb 09, 2019	completed



```
polyspace-access -generate-migration-commands
polyspace-access -migrate
```



Requirements for Migration

The transfer of results from the Metrics repository to the Polyspace Access database requires the `polyspace-access` binary. This binary is available under the `polyspaceroot/polyspace/bin` folder with a Polyspace installation. `polyspaceroot` is the Polyspace product installation folder, for instance `C:\Program Files\Polyspace Server\2019a`.

For syntax and examples, see `polyspace-access`.

Migration of Results

To migrate results from Polyspace Metrics to Polyspace Access, follow these steps. You must be logged in to your Metrics server to complete this operation.

- 1 Identify the Metrics results repository location. The Polyspace Metrics results are stored in the `results-repository` folder at that location.

To view the path to this location, from the desktop interface, go to **Tools > Metrics Server Settings**. Or, at the command line, run the command `psqueue-check-config`.

By default, results are stored under `C:\Users\username\AppData\Roaming\Polyspace_RLDatas\results-repository` on Windows and `/home/username/.polyspace/results-repository` on Linux. *username* is your computer login user name.

- 2 Generate migration scripts.

Once you identify the folder of the repository from which you want to transfer results, define a migration strategy. You can choose to transfer all your projects or you can narrow down the scope of the transfer to a specific set of projects.

Specify a set of projects with the options listed in this table.

Option	Description
<code>-max-project-runs</code> <i>int</i>	Number of most recent analysis runs you want to migrate for each project. For instance, to migrate only the last two analysis runs of a project, specify 2.
<code>-project-date-after</code> <i>YYYY[-MM[-DD]]</i>	Only migrate results that were uploaded to Polyspace Metrics on or after the specified date.
<code>-product</code> <i>productName</i>	Product used to analyze and produce project findings, specified as <code>bug-finder</code> , <code>code-prover</code> , or <code>polyspace-ada</code> .
<code>-analysis-mode</code> <i>mode</i>	Analysis mode used to generate project findings, specified as <code>integration</code> or <code>unit-by-unit</code> .

For example, to transfer only Polyspace Bug Finder analysis results that you uploaded to Polyspace Metrics on or after June 2017, use this command:

```
polyspace-access -generate-migration-commands ^
C:\Users\username\AppData\Roaming\Polyspace_RLDatas\results-repository ^
-output-folder-path C:\Polyspace_Workspace\Migrate^
-project-date-after 2017-06^
-product bug-finder
```

The command outputs a migration script file for each project stored in `C:\Users\username\AppData\Roaming\Polyspace_RLDatas\results-repository` that matches the specified product and date. The migration scripts are stored under `C:\Polyspace_Workspace\Migrate`.

Before you continue, you can optionally open the migration scripts in a text editor and modify the `-project` or `-parent-project` parameters. The parameters correspond to the name of the project and the folder under which it is stored in Polyspace Access, respectively.

3 Migrate the projects.

After you generate the migration scripts, to transfer all the selected projects use those scripts with this migration command :

```
polyspace-access -host hostName -port port ^  
-migrate -option-file-path ^  
C:\Polyspace_Workspace\Migrate
```

The command looks for migration scripts under `C:\Polyspace_Workspace\Migrate` and uploads the results to the Polyspace Access instance that you specify with *hostName*. Enter your Polyspace Access user name and password at the prompt.

hostName and *port* correspond to the host name and port number you specify in the URL of the Polyspace Access interface, for example `https://hostName:port/metrics/index.html`. If you are unsure about which host name and port number to use, contact your Polyspace Access administrator. Depending on your configuration, you might also need to specify the `-protocol` option in the migration command.

During the execution of a migration script, the command generates a temporary `STARTED` file. After each successful execution of a migration script, the command deletes the `STARTED` file and generates a corresponding `DONE` file in the same folder as the script. For example, the command generates `foo.started` during the execution of `foo.cmd`, and then `foo.done` once `foo.cmd` is done. Do not delete these `DONE` files until you have completed your migration from Metrics to Access.

Depending on the amount of data that you are transferring and on your network configuration, the migration might take a long time. You can interrupt the transfer, and then continue from where you left off at a later time. To stop the transfer, press **CTRL+C**. To restart the transfer:

- a Go to the folder where you store the migration scripts and open the `STARTED` file in a text editor. The file might be in a subfolder of the migration scripts folder.
- b Follow the instructions in the file, then reuse the same migration command that you used when you started the migration. The command skips projects that uploaded successfully.

If a project migration fails, go to the migration script folder. See the `FAILED` file for more information.

Differences in SQO Between Polyspace Metrics and Polyspace Access

After you migrate your projects from Polyspace Metrics to Polyspace Access, you might notice differences when you examine your code quality against “Evaluate Polyspace Code Prover Results Against Software Quality Objectives” (Polyspace Code Prover).

The difference is due to the way Polyspace Metrics and Polyspace Access calculate the thresholds for the quality objectives. Polyspace Metrics looks at individual files to determine whether your code achieves a given SQO threshold. For instance, if file `foo.c` does not achieve threshold `SQ02`, then the whole project does not achieve that threshold.

Polyspace Access looks at the whole project to determine whether your source code meets a given SQO threshold. Even if file `foo.c` does not achieve the threshold, the whole project can still meet the specified quality objective threshold.

See Also

More About

- “Register Polyspace Desktop User Interface”
- “Upload Results to Polyspace Access” on page 2-25

Troubleshooting

Troubleshooting in Polyspace Bug Finder

- “Fix License Error -4,0 When Running Polyspace” on page 32-3
- “View Error Information When Analysis Stops” on page 32-4
- “Contact Technical Support About Issues with Running Polyspace” on page 32-6
- “Resolve Error: No Compilation Unit Detected in Your Build” on page 32-9
- “Create Polyspace Projects from Build Systems That Use Unsupported Compilers” on page 32-11
- “Fix Slow Build Process When Polyspace Traces Build” on page 32-17
- “Check if Polyspace Supports Build Scripts” on page 32-18
- “Troubleshoot Project Creation from MinGW Build” on page 32-19
- “Troubleshoot Project Creation from Visual Studio Build” on page 32-20
- “Fix Error: Polyspace Cannot Find Server” on page 32-21
- “Fix Error: Job Manager Cannot Write to Database” on page 32-22
- “Fix Polyspace Compilation Errors About Undefined Identifiers” on page 32-23
- “Fix Polyspace Compilation Errors About Unknown Function Prototype” on page 32-26
- “Fix Polyspace Compilation Errors Related to #error Directive” on page 32-27
- “Fix Polyspace Compilation Errors About Large Objects” on page 32-28
- “Fix Polyspace Compilation Errors Related to Generic Compiler” on page 32-30
- “Fix Polyspace Compilation Errors Related to GNU Compiler” on page 32-31
- “Fix Polyspace Compilation Errors Related to Visual Compilers” on page 32-32
- “Fix Polyspace Compilation Errors Related to Keil or IAR Compiler” on page 32-34
- “Fix Polyspace Compilation Errors Related to Diab Compiler” on page 32-35
- “Fix Polyspace Compilation Errors Related to Green Hills Compiler” on page 32-37
- “Fix Polyspace Compilation Errors Related to TASKING Compiler” on page 32-39
- “Fix Polyspace Compilation Errors Related to Texas Instruments Compilers” on page 32-41
- “Fix Errors from Use of Polyspace Header Files” on page 32-42
- “Fix Polyspace Compilation Errors About Namespace std Without Prefix” on page 32-44
- “Fix Polyspace Compilation Warnings Related to Assertion or Memory Allocation Functions” on page 32-45
- “Fix Polyspace Compilation Errors About In-Class Initialization (C++)” on page 32-46
- “Update Eclipse Java Version for Polyspace Plug-in” on page 32-47
- “Fix MATLAB Crashes Referring to Polyspace in matlabroot” on page 32-48
- “Diagnose Why Coding Standard Violations Do Not Appear as Expected” on page 32-49
- “Check Why a Bug Finder Defect Does Not Appear as Expected” on page 32-52
- “Fix Insufficient Memory Errors During Polyspace Report Generation” on page 32-55

- “Fix Errors or Slow Polyspace Runs from Disk Defragmentation and Anti-virus Software” on page 32-58
- “Fix SQLite I/O Errors on Running Polyspace” on page 32-60
- “Fix Polyspace Errors Related to Temporary Files” on page 32-61
- “Fix Errors Applying Custom Annotation Format for Polyspace Results” on page 32-63
- “Fix Issues When when Integrating Polyspace with MATLAB and Simulink” on page 32-65
- “Check Why Polyspace Functions are Unavailable in MATLAB” on page 32-67
- “Troubleshoot Java Incompatibility in Polyspace Plugin for Eclipse” on page 32-68

Fix License Error -4,0 When Running Polyspace

Issue

When you try to run Polyspace, you get this error message:

```
License Error -4,0
```

Possible Cause: Another Polyspace Instance Running

You can open multiple instances of Polyspace, but you can only run one code analysis at a time.

If you try to run Polyspace processes from multiple windows, you will get a License Error -4,0 error.

Solution

Only run one analysis at a time, including any command-line or plugin analyses.

Possible Cause: Prior Polyspace Run in Simulink or MATLAB Coder

If you run Polyspace on generated code in the Simulink user interface or in the MATLAB Coder app, you can get a license error if you try to run a subsequent analysis in the Polyspace user interface. You get the error even if the previous run is over.

Solution

Run the subsequent analysis using the method that you used before, that is, in the Simulink user interface or MATLAB Coder app.

If you want to run the analysis in the Polyspace user interface, close Simulink or MATLAB Coder and then rerun the analysis.




View Error Information When Analysis Stops

If the analysis stops, you can view error information on the screen, either in the user interface or at the command-line terminal. Alternatively, you can view error information in a log file generated during analysis. Based on the error information, you can either fix your source code, add missing files or change analysis options to get past the error.

View Error Information in User Interface

- 1 View the errors on the **Output Summary** tab.

The messages on this tab appear with the following icons.

Icon	Meaning
	Error that blocks analysis. For instance, the analysis cannot find a variable declaration or definition and therefore cannot determine the variable type.
	Warning about an issue that does not block analysis by itself, but could be related to a blocking error. For instance, the analysis cannot find an include file that is <code>#include-d</code> in your code. The issue does not block the analysis by itself, but if the include file contains the definition of a variable that you use in your source code, you can face an error later.
	Additional information about the analysis.

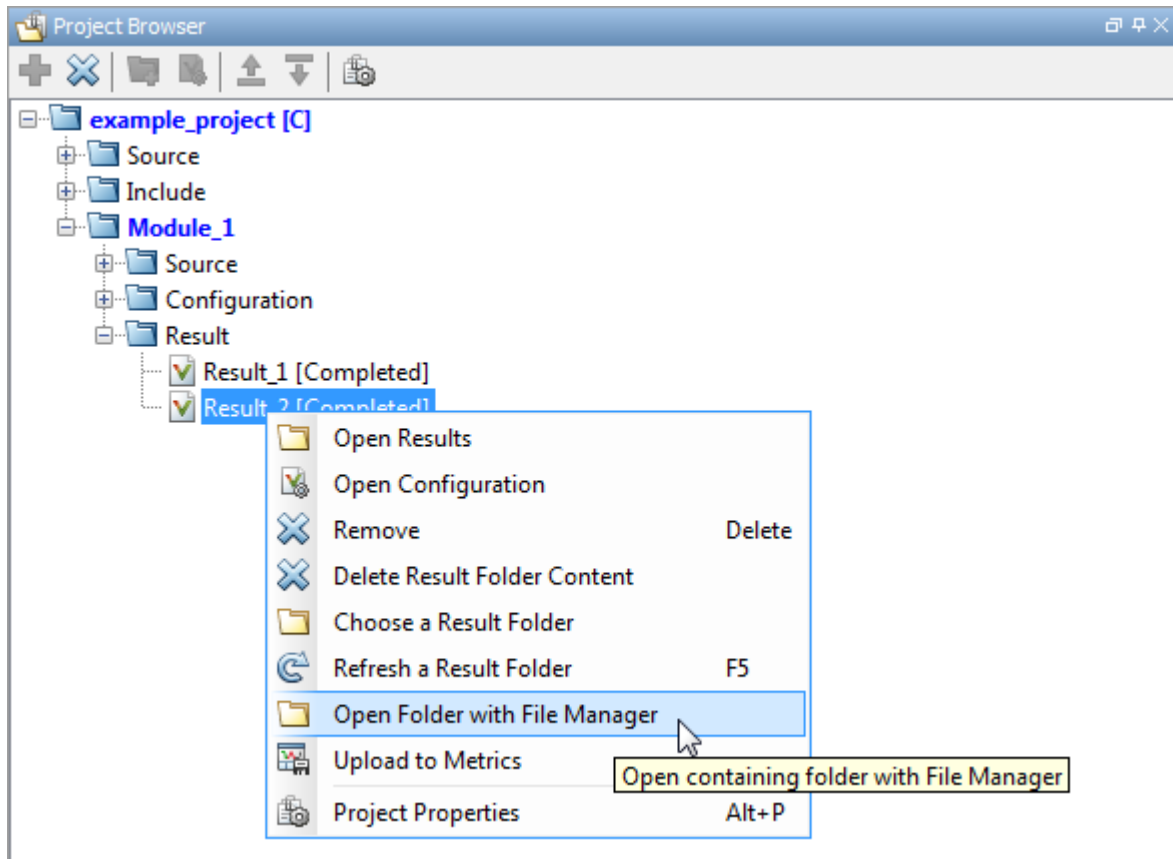
- 2 To diagnose and fix each error, you can do the following:
 - To see further details about the error, select the error message. The details appear in a **Detail** window below the **Output Summary** tab.
 - To open the source code at the line containing the error, double-click the message.

Tip To search the error messages for a specific term, on the **Search** pane, enter your search term. From the drop down list on this pane, select **Output Summary** or **Run Log**. If the **Search** pane is not open by default, select **Windows > Show/Hide View > Search**.

View Error Information in Log File

You can view errors directly in the log file. The log file is in your results folder. To open the log file:

- 1 Right-click the result folder name on the **Project Browser** pane. From the context menu, select **Open Folder with File Manager**.



- 2 Open the log file, `Polyspace_R20##n_ProjectName_date-time.log`
- 3 To view the errors, scroll through the log file, starting at the end and working backward.

The following example shows sample log file information. The error has occurred because a variable `var` used in the code is not defined earlier.

```
C:\missing_include.c, line 4: error: identifier "var" is undefined
|   var = func();
|   ^
```

```
1 error detected in the compilation of "missing_include.c".
C:\missing_include.c: warning: Failed compilation.
Global compilation phase...
```

See Also

Stop analysis if a file does not compile (`-stop-if-compile-error`) | File does not compile

Contact Technical Support About Issues with Running Polyspace

To contact MathWorks Technical Support, use this page. You need a MathWorks Account login and password. For faster turnaround with an issue in Polyspace, besides the required system information, provide appropriate code that reproduces the issue or the verification log file.

Provide System Information

When you enter a support request, provide the following system information:

- Hardware configuration
- Operating system
- Polyspace and MATLAB license numbers
- Specific version numbers for Polyspace products
- Installed Bug Report patches

To obtain your configuration information, do one of the following:

- In the Polyspace user interface, select **Help > About**.
- At the command line, navigate to your Polyspace installation folder, for instance `C:\Program Files\Polyspace\R2023a` (Windows) or `/usr/local/Polyspace/R2023a` (UNIX), and run the command that corresponds to your product and platform:

Product	Command
Polyspace Bug Finder Polyspace Code Prover	<ul style="list-style-type: none"> • UNIX <code>polyspace/bin/polyspace-bug-finder -ver</code> • Windows <code>polyspace\bin\polyspace-bug-finder.exe -ver</code>
Polyspace Bug Finder Server Polyspace Code Prover Server	<ul style="list-style-type: none"> • UNIX <code>polyspace/bin/polyspace-bug-finder-server -ver</code> • Windows <code>polyspace\bin\polyspace-bug-finder-server.exe -ver</code>

If you configure Polyspace to offload the analysis from a client machine to a server machine, to obtain the system configuration of the server machine from the client machine, add options `-batch -scheduler MJSName@host` to the command. For example:

```
polyspace/bin/polyspace-bug-finder -ver -batch -scheduler MJSName@host
```

Here, *MJSName* is the name of the MATLAB Job Scheduler on the head node of the MATLAB Parallel Server cluster and *host* is the host name of the server machine that hosts the head node of this cluster.

Provide Information About the Issue

Depending on the issue, provide appropriate artifacts to help Technical Support understand and reproduce the issue.

Compilation Errors

If you face compilation issues with your project, see “Troubleshoot Compilation Errors”. If you are still having issues, contact technical support with the following information:

- The analysis log.

The analysis log is a text file generated in your results folder and titled `Polyspace_version_project_date_time.log`. It contains the error message, the options used for the analysis and other relevant information.

- The source files related to the compilation error or the complete results folder if possible.

If you cannot provide the source files:

- Try to provide a screenshot of the source code section that causes the compilation issue.
- Try to reproduce the issue with a different code. Provide that code to technical support.

Polyspace as You Code writes the contents of compilation error messages to a log file. The log is generated in your results folder and titled `polyspace_err.log`. Provide this log to technical support if you encounter a compilation issue with Polyspace as You Code.

Errors in Project Creation from Build Systems

If you face errors in creating a project from your build system, see “Troubleshoot Project Creation”.

If you are still having issues, contact technical support with debug information. To provide the debug information:

- 1 Run `polyspace-configure` at the command line with the option `-easy-debug`. For instance:

```
polyspace-configure options -easy-debug pathToFolder buildCommand
```

Here:

- `options` is the list of `polyspace-configure` options that you typically use.
- `buildCommand` is the build command that you use, for instance, `make`.
- `pathToFolder` is the folder where you want to store debug information, for instance, `C:\Temp\BuildLogs`. After a `polyspace-configure` run, the path provided contains a zipped file ending with `pscfg-output.zip`. The zipped file contains debug information only and does not contain source files traced in the build.

Make sure that you do not use the option `-verbose` or `-silent` after `-easy-debug`. These options reduce or modify the information logged and might make debugging difficult.

- 2 Send this zipped file ending with `pscfg-output.zip` to MathWorks Technical Support for further debugging.

You can also create the zipped file with debug information during every `polyspace-configure` run by creating an environment variable `PS_CONFIGURE_OPTIONS` and setting its value to:

`-easy-debug pathToFolder`

where `pathToFolder` is the folder where you want to store debug information.

Verification Result

If you are having trouble understanding a result, see “Complete List of Polyspace Bug Finder Results”.

If you are still having trouble understanding the result, contact technical support with the following information:

- The analysis log.

The analysis log is a text file generated in your results folder and titled `Polyspace_version_project_date_time.log`. It contains the options used for the analysis and other relevant information.

- The source files related to the result or the complete results folder if possible.

If you cannot provide the source files:

- Try to provide a screenshot of the relevant source code from the **Source** pane on the Polyspace user interface.
- Try to reproduce the problem with a different code. Provide that code to technical support.

Provide Polyspace Analysis Statistics File (Optional)

Depending on your issue, you might be asked to provide the `.stats.zip` file. The file is located in your results folder and contains statistics about your analysis, such as options used, time taken by the different phases of the analysis, and the memory consumed by the different processes that ran during the analysis. The file contains no identifying information about your code.

See Also

Related Examples

- “Contact Technical Support About Polyspace Access Issues” on page 33-5

Resolve Error: No Compilation Unit Detected in Your Build

Issue

You can automatically create a Polyspace project or options file by running `polyspace-configure` on your build command at the command line or in the Polyspace user interface. `polyspace-configure` executes your build command, tracks the processes executed, extracts the compiler command invocations from the tracked processes, and determines source files to add to a Polyspace project. If your compiler command does not execute during the build or cannot be found, you might see this error:

```
No compilation unit detected in your build.
```

The error indicates that `polyspace-configure` could not find source files to add to a Polyspace project or options file.

Possible Solutions

Check for Incremental Build

Most build systems perform an incremental build by default. The build commands rebuild only the sources that changed since the last rebuild. For instance, if you build your sources by using `make` or its equivalent, the command rebuilds only those targets in the makefile that are out-of-date. If you run `polyspace-configure` on a build command and none of your sources have changed since the last rebuild, your compiler is not invoked and you see the preceding error. Even if some of the sources have changed, your compiler might rebuild only those sources, leaving your Polyspace project incomplete.

To avoid the error, when running `polyspace-configure`, perform a full or clean build of your sources. For instance, when building by using `make`, you can use the flag `-B` or `--always-make` to rebuild all targets in the makefile.

Check for Compiler Caching

Using a compiler cache is equivalent to performing an incremental build. Compiler caches speed up compilation by caching results of previous compilations. If the same compilation is repeated, the cached results are used instead of a fresh compilation. In this case, `polyspace-configure` cannot detect a compilation because the actual compiler commands are not invoked.

To work around the error, disable any compiler cache that you might be using just before running `polyspace-configure` on your build command. You can reenableView the caching immediately after running `polyspace-configure`. For instance, if you use Ccache, you can disable the caching on the current shell by entering:

```
export CCACHE_DISABLE=1
```

Check for Antivirus Software

The `polyspace-configure` command works by tracking the processes executed by your build command. Certain antivirus software can block this tracking. `polyspace-configure` keeps a known list of antivirus software that block tracking and shows a warning if any of the software is detected. You might be using an antivirus software outside this known list and not see the warning but see a later error instead.

To avoid the issue, temporarily disable your antivirus software when running `polyspace-configure`. Some antivirus software allow you to specify a list of processes that must not be blocked. You might be able to work around the issue by specifying Polyspace processes in your allowlist. For details, see “Fix Errors or Slow Polyspace Runs from Disk Defragmentation and Anti-virus Software” on page 32-58.

Check for Unsupported Compilers

The `polyspace-configure` command supports the same compilers as Polyspace. Check if your compiler is supported in `Compiler (-compiler)`. For each supported compiler, `polyspace-configure` can recognize a known set of compiler invocation commands. If your compiler is not supported or is supported but uses an invocation command outside this known set, `polyspace-configure` fails to recognize the compiler invocation and produces the preceding error.

If your compiler is supported but uses a nonstandard invocation command, or is closely related to a supported compiler, you might be able to extend support to your compiler command. See “Create Polyspace Projects from Build Systems That Use Unsupported Compilers” on page 32-11. In all other cases, contact MathWorks Technical Support. See “Contact Technical Support About Issues with Running Polyspace” on page 32-6.

Check for Source Exclusions

You can use the option `-include-sources` or `-exclude-sources` with the `polyspace-configure` command to include or exclude certain sources from the generated Polyspace project or options file. If you use these options, make sure to check their arguments and ensure that you have not accidentally excluded more source files than you intend.

See also “Select Files for Polyspace Analysis Using Pattern Matching” on page 4-11.

See Also

`polyspace-configure`

Related Examples

- “Requirements for Project Creation from Build Systems” on page 13-24
- “Create Polyspace Projects from Build Systems That Use Unsupported Compilers” on page 32-11

Create Polyspace Projects from Build Systems That Use Unsupported Compilers

Issue

Your compiler is not supported for automatic project creation from build commands.

Cause

For automatic project creation from your build system, your compiler configuration must be available to Polyspace. Polyspace provides a compiler configuration file only for certain compilers.

For information on which compilers are supported, see “Requirements for Project Creation from Build Systems” on page 13-24.

Solution

To enable automatic project creation for an unsupported compiler, you can write your own compiler configuration file.

- 1 Copy one of the existing configuration files from *polyspaceroot\polyspace\configure\compiler_configuration*. Select the configuration that most closely corresponds to your compiler using the mapping between the configuration files and compiler names on page 32-15.
- 2 Save the file as *my_compiler.xml*. *my_compiler* can be a name that helps you identify the file.

To edit the file, save it outside the installation folder. After you have finished editing, you must copy the file back to *polyspaceroot\polyspace\configure\compiler_configuration*.

- 3 Edit the contents of the file to represent your compiler. Replace the entries between the XML elements with appropriate content.
- 4 After saving the edited XML file to *polyspaceroot\polyspace\configure\compiler_configuration*, create a project automatically using your build command.

If you see errors, for instance, compilation errors, contact MathWorks Technical Support. After tracing your build command, the software compiles certain files using the compiler specifications detected from your configuration file and build command. Compilation errors might indicate issues in the configuration file.

Tip To quickly see if your compiler configuration file works, run the automatic project setup on a sample build that does not take much time to complete. After you have set up a project with your compiler configuration file, you can use this file for larger builds.

Elements of Compiler Configuration File

The following table lists the XML elements in the compiler configuration file file with a description of what the content within the element represents.

XML Element	Content Description	Content Example for GNU C Compiler
<pre><compiler_names><name> ... </name></compiler_names></pre>	<p>Name of the compiler executable. This executable transforms your .c files into object files. You can add several binary names, each in a separate <name>...</name> element. The software checks for each of the provided names and uses the compiler name for which it finds a match.</p> <p>You must not specify the linker binary inside the <name>...</name> elements.</p> <p>If the name that you specify is present in an existing compiler configuration file, an error occurs. To avoid the error, use the additional option <code>-compiler-config my_compiler.xml</code> when tracing the build so that the software explicitly uses your compiler configuration file.</p>	<ul style="list-style-type: none"> • gcc • gpp
<pre><include_options><opt> ... </opt></include_options></pre>	<p>The option that you use with your compiler to specify include folders.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-I</p>
<pre><system_include_options> <opt> ... </opt> </system_include_options></pre>	<p>The option that you use with your compiler to specify system headers.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-isystem</p>
<pre><preinclude_options><opt> ... </opt></preinclude_options></pre>	<p>The option that you use with your compiler to force inclusion of a file in the compiled object.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-include</p>

XML Element	Content Description	Content Example for GNU C Compiler
<pre><define_options><opt> ... </opt></define_options></pre>	<p>The option that you use with your compiler to predefine a macro.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-D</p>
<pre><undefine_options><opt> ... </opt></undefine_options></pre>	<p>The option that you use with your compiler to undo any previous definition of a macro.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-U</p>
<pre><semantic_options><opt> ... </opt></semantic_options></pre>	<p>The options that you use to modify the compiler behavior. These options specify the language settings to which the code must conform.</p> <p>You can use the <code>isPrefix</code> attribute to specify multiple options that have the same prefix and the <code>numArgs</code> attribute to specify options with multiple arguments. For instance:</p> <ul style="list-style-type: none"> • Instead of <pre><opt>-m32</opt> <opt>-m64</opt></pre> <p>You can write <code><opt isPrefix="true">-m</opt></code>.</p> • Instead of <pre><opt>-std=c90</opt> <opt>-std=c99</opt></pre> <p>You can write <code><opt numArgs="1">-std</opt></code>. If your makefile uses <code>-std=c90</code> instead of <code>-std=c90</code>, this notation also supports that usage.</p> 	<ul style="list-style-type: none"> • -ansi • -std =C90 • -std =c++11 • -fun signed -char

XML Element	Content Description	Content Example for GNU C Compiler
<pre><compiler> ... </compiler></pre>	<p>The Polyspace compiler option that corresponds to or closely matches your compiler. The content of this element directly translates to the option Compiler in your Polyspace project or options file.</p> <p>For the complete list of compilers available, see <code>Compiler (-compiler)</code>.</p>	<p>gnu4.7</p>
<pre><preprocess_options_list> <opt> ... </opt> </preprocess_options_list></pre>	<p>The options that specify how your compiler generates a preprocessed file.</p> <p>You can use the macro <code>\$(OUTPUT_FILE)</code> if your compiler does not allow sending the preprocessed file to the standard output. Instead it defines the preprocessed file internally.</p>	<p>-E</p> <p>For an example of the <code>\$(OUTPUT_FILE)</code> macro, see the existing compiler configuration file <code>cl2000.xml</code>.</p>
<pre><preprocessed_output_file> ... </preprocessed_output_file></pre>	<p>The name of file where the preprocessed output is stored.</p> <p>You can use the following macros when the name of the preprocessed output file is adapted from the source file:</p> <ul style="list-style-type: none"> • <code>\$(SOURCE_FILE)</code>: Source file name • <code>\$(SOURCE_FILE_EXT)</code>: Source file extension • <code>\$(SOURCE_FILE_NO_EXT)</code>: Source file name without extension <p>For instance, use <code>\$(SOURCE_FILE_NO_EXT).pre</code> when the preprocessor file name has the same name as the source file, but with extension <code>.pre</code>.</p>	<p>For an example of this element, see the existing compiler configuration file <code>xc8.xml</code>.</p>
<pre><src_extensions><ext> ... </ext></src_extensions></pre>	<p>The file extensions for source files.</p>	<ul style="list-style-type: none"> • c • cpp • c++

XML Element	Content Description	Content Example for GNU C Compiler
<pre><obj_extensions><ext> ... </ext></obj_extensions></pre>	The file extensions for object files.	
<pre><precompiled_header_extensions> ... </precompiled_header_extensions></pre>	The file extensions for precompiled headers (if available).	
<pre><polyspace_extra_options_list> <opt> ... </opt> <opt> ... </opt> </polyspace_extra_options_list></pre>	<p>Additional options that are used for the subsequent analysis.</p> <p>For instance, to avoid compilation errors in the subsequent analysis due to non-ANSI extension keywords, enter <code>-D keyword=value</code>, for example:</p> <pre><polyspace_extra_options_list> <opt>-D MACR01</opt> <opt>-D MACR02=VALUE</opt> </polyspace_extra_options_list></pre> <p>For more information, see Preprocessor definitions (-D).</p>	

Mapping Between Existing Configuration Files and Compiler Names

Select the configuration file in `polyspaceroot\polyspace\configure\compiler_configuration\` that most closely resembles the configuration of your compiler. Use the following table to map compilers to their configuration files.

Compiler Name	Vendor	XML File
ARM	ARM Keil	armcc.xml
		armclang.xml
Visual C++	Microsoft	cl.xml
Clang	Not applicable	clang.xml
CodeWarrior	NXP	cw_ppc.xml
		cw_s12z.xml
cx6808	Cosmic	cx6808.xml
		cosmic.xml
Diab	Wind River	diab.xml
gcc	Not applicable	gcc.xml
Green Hills	Green Hills Software	ghs_arm.xml
		ghs_arm64.xml
		ghs_i386.xml

Compiler Name	Vendor	XML File
		ghs_ppc.xml
		ghs_rh850.xml
		ghs_tricore.xml
IAR Embedded Workbench	IAR	iar.xml
		iar-arm.xml
		iar-avr.xml
		iar-msp430.xml
		iar-rh850.xml
		iar-riscv.xml
		iar-rl78.xml
Renesas	Renesas	renesas-rh850.xml
		renesas-rl78.xml
		renesas-rx.xml
		renesas-sh.xml
TASKING®	Altium	tasking.xml
		tasking-166.xml
		tasking-850.xml
		tasking-arm.xml
Tiny C	Not applicable	tcc.xml
TM320 and its derivatives	Texas Instruments	ti_arm.xml
		ti_c28x.xml
		ti_c6000.xml
		ti_msp430.xml
xc8 (PIC)	Microchip	microchip.xml

Fix Slow Build Process When Polyspace Traces Build

Issue

In some cases, your build process can run slower when Polyspace traces the build.

Cause

Polyspace caches information in files stored in the system temporary folder, such as `C:\Users\%User_Name%\AppData\Local\Temp`, in Windows. Your build can take a long time to perform read/write operations to this folder. Therefore, the overall build process is slow.

Solution

You can work around the slow build process by changing the location where Polyspace stores cache information. For instance, you can use a cache path local to the drive from which you run build tracing. To create and use a local folder `ps_cache` for storing cache information, use the advanced option `-cache-path ./ps_cache`.

- If you trace your build from the Polyspace user interface, enter this flag in the field **Add advanced configure options**.
- If you trace your build from the DOS/ UNIX or MATLAB command line, use this flag with the `polyspace-configure` command.

For more information, see `polyspace-configure`.

Check if Polyspace Supports Build Scripts

Issue

This topic is relevant only if you are creating a Polyspace project in Windows from your build scripts.

When Polyspace traces your build script in a Windows console application other than `cmd.exe`, the command fails. However, the build command by itself executes to completion.

For instance, your build script executes to completion from the Cygwin shell. However, when Polyspace traces the build, the build script throws an error.

Possible Cause

When you launch a Windows console application, your environment variables are appropriately set. Alternate console applications such as the Cygwin shell can set your environment differently from `cmd.exe`.

Polyspace attempts to trace your build script with the assumption that the script runs to completion in `cmd.exe`. Therefore, even if your script runs to completion in the alternate console application, when Polyspace traces the build, the script can fail.

Solution

Make sure that your build script executes to completion in the `cmd.exe` interface. If the build executes successfully, create a wrapper `.bat` file around your script and trace this file.

For instance, before you trace a build command that executes to completion in the Cygwin shell, launch the Cygwin shell from `cmd.exe` and then run your build script. For example, if you use a script `build.sh` to build your code, enter the following command at the DOS command line:

```
cmd.exe /C "C:\cygwin64\bin\bash.exe" --login -c build.sh
```

If the steps do not execute to completion, Polyspace cannot trace your build.

If the steps complete successfully, trace the build command after launching it from `cmd.exe`. For instance, on the command-line, do the following to create a Polyspace options file.

- 1 Enter your build commands in a `.bat` file.

```
rem @echo off
cmd.exe /C "C:\cygwin64\bin\bash.exe" --login -c build.sh
```

Name the file, for instance, `launching.bat`.

- 2 Trace the build commands in the `.bat` file and create a Polyspace options file.

```
"C:\Program Files\MATLAB\R2023a\polyspace\bin\polyspace-configure.exe"
-output-options-file myOptions.txt launching.bat
```

You can now run `polyspace-bug-finder` or `polyspace-bug-finder-server` on the options file.

Troubleshoot Project Creation from MinGW Build

Issue

You create a project from a MinGW build, but get an error when running an analysis on the project. The error message comes from using one of these keywords: `__declspec`, `__cdecl`, `__fastcall`, `__thiscall` or `__stdcall`.

Cause

When you create a project from a MinGW build, the project uses a GNU compiler. Polyspace does not recognize these keywords for the GNU compilers.

Solution

Replace these keywords with equivalent keywords just for the purposes of analysis.

Before analysis, for the option Preprocessor definitions (-D), enter:

- `__declspec(x)=__attribute__((x))`
- `__cdecl=__attribute__((__cdecl__))`
- `__fastcall=__attribute__((__fastcall__))`
- `__thiscall=__attribute__((__thiscall__))`
- `__stdcall=__attribute__((__stdcall__))`

If you are running Polyspace on the command line in a UNIX shell, add double quotes around the -D option. For instance, use:

```
"-D __cdecl=__attribute__((__cdecl__))"
```

Troubleshoot Project Creation from Visual Studio Build

You can run the `polyspace-configure` command to create a Polyspace project or options file from a Visual Studio build. The command monitors the processes executed during the build and extracts information required for the project or options file

You can trace your Visual Studio build in one of the following ways:

- Build your Visual Studio project completely at the command line with `msbuild` while tracing this build with `polyspace-configure`.

In this workflow, you run `polyspace-configure` on an `msbuild` command with a Visual Studio project (`.vcxproj`) file. For instance, in a Visual Studio 2019 developer prompt, enter the following:

```
polyspace-configure msbuild TestProject.vcxproj /t:Rebuild
```

- Build your Visual Studio project in the Visual Studio IDE while tracing this build with `polyspace-configure`.

Run `polyspace-configure` on the `devenv.exe` executable to open the Visual Studio IDE, build your project or solution within the IDE, and then close the IDE.

See “Create Polyspace Projects from Visual Studio Build” on page 2-9.

If running `polyspace-configure` on the `msbuild` command does not work properly, do the following:

- 1 Stop the `msbuild` process.
- 2 Set the environment variable `MSBUILDDISABLENODEREUSE` to 1.
- 3 Restart `polyspace-configure` on `msbuild`, this time using the `/nodereuse:false` option. For instance:

```
polyspace-configure msbuild TestProject.vcxproj /t:Rebuild /nodereuse:false
```

See Also

`polyspace-configure`

Fix Error: Polyspace Cannot Find Server

Message

```
Error: Cannot instantiate Polyspace cluster
| Check the -scheduler option validity or your default cluster profile
| Could not contact an MJS lookup service using the host computer_name.
  The hostname, computer_name, could not be resolved.
```

Possible Cause

Polyspace uses information provided in the preferences of a Polyspace desktop product to locate the server. If this information is incorrect, the software cannot locate the server.

Solution

Open the user interface of the Polyspace desktop product. Check if the server information provided is correct.

- 1 Select **Tools > Preferences**.
- 2 Select the **Server Configuration** tab. Check your server information.

For instance, the entry in **Job scheduler host name** must match the host name of the computer that forms the head node of the MATLAB Parallel Server cluster. For more information, see “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

Fix Error: Job Manager Cannot Write to Database

Message

Unable to write data to the job manager database

Possible Cause

If the computer that forms the head node of the MATLAB Parallel Server cluster cannot send data to the client computer, you see this error. The most likely reasons for the remote computer being unable to connect to the client computer are:

- Firewalls do not allow traffic from the MATLAB Job Scheduler to the client.
- The MATLAB Job Scheduler cannot resolve the short hostname of the client computer.

Workaround

Add localhost IP to configuration.

- 1 In the user interface of the Polyspace desktop products, select **Tools > Preferences**.
- 2 On the **Server Configuration** tab, in the **Localhost IP address** field, enter the IP address of your local computer.

To retrieve your IP address:

- Windows
 - 1 Open **Control Panel > Network and Sharing Center**.
 - 2 Select your active network.
 - 3 In the Status window, click **Details**. Your IP address is listed under **IPv4 address**.
- Linux — Run the `ifconfig` command and find the `inet addr` corresponding to your network connection.
- Mac — Open **System Preferences > Network**.

See Also

Related Examples

- “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”
- “Connection Problems Between the Client and MATLAB Job Scheduler” (Parallel Computing Toolbox)

Fix Polyspace Compilation Errors About Undefined Identifiers

Issue

Polyspace verification fails during the compilation phase with a message about undefined identifiers.

The message indicates that Polyspace cannot find a variable definition. Therefore, it cannot identify the variable type.

Possible Cause: Missing Files

The source code you provided does not contain the variable definition. For instance, the variable is defined in an include file that Polyspace cannot find.

If you `#include-d` the include file in your source code but did not add it to your Polyspace project, you see a previous warning:

```
Warning: could not find include file "my_include.h"
```

Solution

If the variable definition occurs in an include file, add the folder that contains the include file.

- In the user interface of the Polyspace desktop products, add the folder to your project.

For more information, see “Add Source Files for Analysis in Polyspace Desktop User Interface” on page 2-2.

- At the command line, use the flag `-I` with the `polyspace-bug-finder` command.

For more information, see `-I`.

Possible Cause: Unrecognized Keyword

The variable represents a keyword that your compiler recognizes but is not part of the ANSI C standard. Therefore, Polyspace does not recognize it.

For instance, some compilers interpret `__SP` as a reference to the stack pointer.

Solution

If the variable represents a keyword that Polyspace does not recognize, replace or remove the keyword from your source code or preprocessed code.

If you remove or replace the keyword from the preprocessed code, you can avoid the compilation error while keeping your source code intact. You can do one of the following:

- Replace or remove each individual unknown keyword using an analysis option. Replace the compiler-specific keyword with an equivalent keyword from the ANSI C Standard.

For information on the analysis option, see `Preprocessor definitions (-D)`.

- Declare the unknown keywords in a separate header file using `#define` directives. Specify that header file using an analysis option.

For information on the analysis option, see `Include (-include)`. For a sample header file, see “Gather Compilation Options Efficiently” on page 13-32.

Possible Cause: Declaration Embedded in `#ifdef` Statements

The variable is declared in a branch of an `#ifdef macro_name` preprocessor directive. For instance, the declaration of a variable `max_power` occurs as follows:

```
#ifdef _WIN32
  #define max_power 31
#endif
```

Your compilation toolchain might consider the macro `macro_name` as implicitly defined and execute the `#ifdef` branch. However, the Polyspace compilation might not consider the macro as defined. Therefore, the `#ifdef` branch is not executed and the variable `max_power` is not declared.

Solution

To work around the compilation error, do one of the following:

- Use **Target & Compiler** options to directly specify your compiler. For instance, to emulate a Visual C++ compiler, set the **Compiler** to `visual12.0`. See “Target and Compiler”.
- Define the macro explicitly using the option `Preprocessor definitions (-D)`.

Note If you create a Polyspace by tracing your build commands, most **Target & Compiler** options are automatically set.

Possible Cause: Project Created from Non-Debug Build

This can be a possible cause only if the undefined identifier occurs in an `assert` statement (or an equivalent Visual C++ macro such as `ASSERT` or `VERIFY`).

Typically, you come across this error in the following way. You create a Polyspace project from a build system in non-debug mode. When you run an analysis on the project, you face a compilation error from an undefined identifier in an `assert` statement. You find that the identifier `my_identifier` is defined in a `#ifndef NDEBUG` statement, for instance as follows:

```
#ifndef NDEBUG
int my_identifier;
#endif
```

The C standard states that when the `NDEBUG` macro is defined, all `assert` statements must be disabled.

Most IDEs define the `NDEBUG` macro in their build systems. When you build your source code in your IDE in non-debug mode, code in a `#ifndef NDEBUG` statement is removed during preprocessing. For instance, in the preceding example, `my_identifier` is not defined. If `my_identifier` occurs only in `assert` statements, it is not used either, because `NDEBUG` disables `assert` statements. You do not have compilation errors from undefined identifiers and your build system executes successfully.

Polyspace does not disable `assert` statements even if `NDEBUG` macro is defined because the software uses `assert` statements internally to enhance verification.

When you create a Polyspace project from your build system, if your build system defines the `NDEBUG` macro, it is also defined for your Polyspace project. Polyspace removes code in a `#ifndef NDEBUG` statement during preprocessing, but does not disable `assert` statements. If `assert` statements in your code rely on the code in a `#ifndef NDEBUG` statement, compilation errors can occur.

In the preceding example:

- The definition of `my_identifier` is removed during preprocessing.
- `assert` statements are not disabled. When `my_identifier` is used in an `assert` statement, you get an error because of undefined identifier `my_identifier`.

Solution

To work around this issue, create a Polyspace project from your build system in debug mode. When you execute your build system in debug mode, `NDEBUG` is not defined. When you create a Polyspace project from this build, `NDEBUG` is not defined for your Polyspace project.

Depending on your project settings, use the option that enables building in debug mode. For instance, if your build system is `gcc`-based, you can define the `DEBUG` macro and undefine `NDEBUG`:

```
gcc -DDEBUG=1 -UNDEBUG *.c
```

Alternatively, you can disable the `assert` statements in your preprocessed code using the option `Preprocessor definitions (-D)`. However, Polyspace will not be able to emulate the `assert` statements.

Fix Polyspace Compilation Errors About Unknown Function Prototype

Issue

During the compilation phase, the software displays a warning or error message about unknown function prototype.

```
the prototype for function 'myfunc' is unknown
```

The message indicates that Polyspace cannot find a function prototype. Therefore, it cannot identify the data types of the function argument and return value, and has to infer them from the calls to the function.

To determine the data types for such functions, Polyspace follows the C99 Standard (ISO/IEC 9899:1999, Chapter 6.5.2.2: Function calls).

- The return type is assumed to be `int`.
- The number and type of arguments are determined by the first call to the function. For instance, if the function takes one `double` argument in the first call, for subsequent calls, the software assumes that it takes one `double` argument. If you pass an `int` argument in a subsequent call, a conversion from `int` to `double` takes place.

During the linking phase, if a mismatch occurs between the number or type of arguments or the return type in different compilation units, the analysis follows an internal algorithm to resolve this mismatch and determine a common prototype.

Cause

The source code you provided does not contain the function prototype. For instance, the function is declared in an include file that Polyspace cannot find.

If you `#include`-d the include file in your source code but did not add it to your Polyspace project, you see a previous warning:

```
Warning: could not find include file "my_include.h"
```

Solution

Search for the function declaration in your source repository.

If you find the function declaration in an include file, add the folder that contains the include file.

- In the user interface of the Polyspace desktop products, add the folder to your project.

For more information, see “Add Source Files for Analysis in Polyspace Desktop User Interface” on page 2-2.

- At the command line, use the flag `-I` with the `polyspace-bug-finder` command.

For more information, see `-I`.

Fix Polyspace Compilation Errors Related to #error Directive

Issue

The analysis stops with a message containing a #error directive. For instance, the following message appears: #error directive: !Unsupported platform; stopping!.

Cause

You typically use the #error directive in your code to trigger a fatal error in case certain macros are not defined. Your compiler implicitly defines the macros, therefore the error is not triggered when you compile your code. However, the default Polyspace compilation does not consider the macros as defined, therefore, the error occurs.

For instance, in the following example, the #error directive is reached only if the macros `__BORLANDC__`, `__VISUALC32__` or `__GNUC__` are not defined. If you use a GNU C compiler, for instance, the compiler considers the macro `__GNUC__` as defined and the error does not occur. However, if you use the default Polyspace compilation, it does not consider the macros as defined.

```
#if defined(__BORLANDC__) || defined(__VISUALC32__)
#define MYINT int
#elif defined(__GNUC__)
#define MYINT long
#else
#error !Unsupported platform; stopping!
#endif
```

Solution

For successful compilation, do one of the following:

- Specify a compiler such as `visual12.0` or `gnu4.9`. Specifying a compiler defines some of the compilation flags for the analysis.

For more information, see `Compiler (-compiler)`.

- If the available compiler options do not match your compiler, explicitly define one of the compilation flags `__BORLANDC__`, `__VISUALC32__`, or `__GNUC__`.

For more information, see `Preprocessor definitions (-D)`.

Fix Polyspace Compilation Errors About Large Objects

Issue

The analysis stops during compilation with a message indicating that an object is too large.

Cause

The error happens when the software detects an object such as an array, union, structure, or class, that is too big for the pointer size of the selected target.

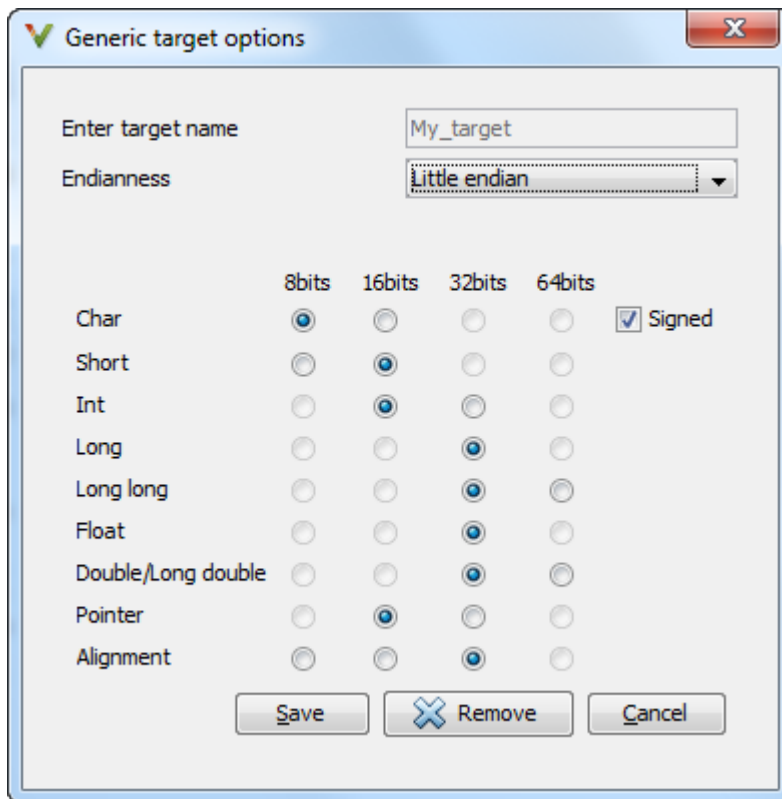
For instance, you get the message, `Limitation: struct or union is too large` in the following example. You specify a pointer size of 16 bits. The maximum object size allocated to a pointer, and therefore the maximum allowed size for an object, can be $2^{16}-1$ bytes. However, you declare a structure as follows:

- ```
struct S
{
 char tab[65536];
}s;
```
- ```
struct S
{
    char tab[65534];
    int val;
}s;
```

Solution

- 1 Check the pointer size that you specified through your target processor type. For more information, see `Target processor type (-target)`.

For instance, in the following, the pointer size for a custom target `My_target` is 16 bits.



- 2 Change your code or specify a different pointer size.

For instance, you can:

- Declare an array of smaller size in the structure.

If you are using a predefined target processor type, the pointer size is likely to be the same as the pointer size on your target architecture. Therefore, your declaration might cause errors on your target architecture.

- Change the pointer size of the target processor type that you specified, if possible.

Otherwise, specify another target processor type with larger pointer size or define your own target processor type. For more information on defining your own processor type, see `Generic target options`.

Note Polyspace imposes an internal limit of 128 MB on the size of data structures. Even if your target processor type specification allows data structures of larger size, this internal limit constrains the data structure sizes.

Fix Polyspace Compilation Errors Related to Generic Compiler

If you use a generic compiler, you can encounter this issue. For more information, see `Compiler (-compiler)`.

Issue

The analysis stops with an error message related to a non-ANSI C keyword, for instance, `data` or attributes such as `__attribute__((weak))`.

Depending on the location of the keyword, the error message can vary. For instance, this line causes the error message: `expected a ";"`.

```
data int tab[10];
```

Cause

The generic Polyspace compiler supports only ANSI C keywords. If you use a language extension, the generic compiler does not recognize it and treats the keyword as a regular identifier.

Solution

Specify your compiler by using the option `Compiler (-compiler)`.

If your compiler is not directly supported or is not based on a supported compiler, you can use the generic compiler. To work around the compilation errors:

- If the keyword is related to memory modelling, remove it from the preprocessed code. For instance, to remove the `data` keyword, enter `data=` for the option `Preprocessor definitions (-D)`.
- If the keyword is related to an attribute, remove attributes from the preprocessed code. Enter `__attribute__(x)=` for the option `Preprocessor definitions (-D)`.

If your code has this line:

```
void __attribute__((weak)) func(void);
```

And you remove attributes, the analysis reads the line as:

```
void func(void);
```

When you use these workarounds, your source code is not altered.

Fix Polyspace Compilation Errors Related to GNU Compiler

If you choose `gnu` for the option `Compiler (-compiler)`, you can encounter this issue.

Issue

The Polyspace analysis stops with a compilation error.

Cause

You are using certain advanced compiler-specific extensions that Polyspace does not support. See “Limitations”.

Solution

For easier portability of your code, avoid using the extensions.

If you want to use the extensions and still analyze your code, wrap the unsupported extensions in a preprocessor directive. For instance:

```
#ifdef POLYSPACE
    // Supported syntax
#else
    // Unsupported syntax
#endif
```

For regular compilation, do not define the macro `POLYSPACE`. For Polyspace analysis, enter `POLYSPACE` for the option `Preprocessor definitions (-D)`.

If the compilation error is related to assembly language code, use the option `-asm-begin -asm-end`.

Fix Polyspace Compilation Errors Related to Visual Compilers

The following messages appear if the compiler is based on a Visual compiler. For more information, see `Compiler (-compiler)`.

Import Folder

When a Visual application uses `#import` directives, the Visual C++ compiler generates a header file with extension `.tlh` that contains some definitions. To avoid compilation errors during Polyspace analysis, you must specify the folder containing those files.

Original code:

```
#include "stdafx.h"
#include <comdef.h>
#import <MsXml.tlb>
MSXML::_xml_error e ;
MSXML::DOMDocument* doc ;
int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

Error message:

```
"../sources/ImportDir.cpp", line 7: catastrophic error: could not
open source file "../MsXml.tlh"
    #import <MsXml.tlb>
```

The Visual C++ compiler generates these files in its “build-in” folder (usually Debug or Release). In order to provide those files:

- Build your Visual C++ application.
- Specify your build folder for the Polyspace analysis.

pragma Pack

Using a different value with the compile flag (`#pragma pack`) can lead to a linking error message.

Original code:

test1.cpp	type.h	test2.cpp
<pre>#pragma pack(4) #include "type.h"</pre>	<pre>struct A { char c ; int i ; } ;</pre>	<pre>#pragma pack(2) #include "type.h"</pre>

Error message:

```
Pre-linking C++ sources ...
"../sources/type.h", line 2: error: declaration of class "A" had
a different meaning during compilation of "test1.cpp"
(class types do not match)
    struct A
```

```
^
  detected during compilation of secondary translation unit
"test2.cpp"
```

To continue the analysis, use the option `Ignore pragma pack directives (-ignore-pragma-pack)`.

C++/CLI

Polyspace does not support Microsoft C++/CLI, a set of language extensions for .NET programming.

You can get errors such as:

```
error: name must be a namespace name
|         using namespace System;
```

Or:

```
error: expected a declaration
|         public ref class Form1 : public System::Windows::Forms::Form
```

Fix Polyspace Compilation Errors Related to Keil or IAR Compiler

If you use the compiler, Keil or IAR, you can encounter this issue. For more information, see `Compiler (-compiler)`.

Missing Identifiers

Issue

The analysis stops with the error message, `expected an identifier`, as if an identifier is missing. However, in your source code, you can see the identifier.

Cause

If you select Keil or IAR as your compiler, the software removes certain keywords during preprocessing. If you use these keywords as identifiers such as variable names, a compilation error occurs.

For a list of keywords that are removed, see “Supported Keil or IAR Language Extensions” on page 13-27.

Solution

Specify that Polyspace must not remove the keywords during preprocessing. Define the macros `__PST_KEIL_NO_KEYWORDS__` or `__PST_IAR_NO_KEYWORDS__`.

For more information, see `Preprocessor definitions (-D)`.

Fix Polyspace Compilation Errors Related to Diab Compiler

If you choose `diab` for the option `Compiler` (`-compiler`), you can encounter this issue.

Issue

During Polyspace analysis, you see an error related to a keyword specific to the Diab compiler. For instance, you see an error related to the `restrict` keyword.

Cause

You typically use a compiler flag to enable the keyword. The Polyspace analysis does not enable these keywords by default. You have to make Polyspace aware of your compiler flags.

The Polyspace analysis does not enable these keywords by default to prevent compilation errors. Another user might not enable the keyword and instead use the keyword name as a regular identifier. If Polyspace treats the identifier as a keyword, a compilation error will occur.

Solution

Use the command-line option `-compiler-parameter` in your Polyspace analysis as follows. You use this command-line option to make Polyspace aware of your compiler flags. In the user interface of the Polyspace desktop products, you can enter the command-line option in the field `Other`. You can enter the option multiple times.

The argument of `-compiler-parameter` depends on the keyword that causes the error. Once you enable the keyword, do not use the keyword name as a regular identifier. For instance, once you enable the keyword `pixel`, do not use `pixel` as a variable name. The statement `int pixel = 1` causes a compilation error.

- `restrict` keyword:

You typically use the compiler flag `-Xlibc-new` or `-Xc-new`. For your Polyspace analysis, use `-compiler-parameter -Xc-new`

The following code will not compile with Polyspace unless you specify the compiler flag.

```
int sscanf(const char *restrict, const char *restrict, ...);
```

- PowerPC AltiVec vector extensions such as the `vector` type qualifier:

You typically use the compiler flag `-tPPCALLAV:.` For your Polyspace analysis, use `-compiler-parameter -tPPCALLAV:`

The following code will not compile with Polyspace unless you specify the compiler flag.

```
vector unsigned char vbyte;
vector bool vbool;
vector pixel vpx;

int main(int argc, char** argv)
{
```

```
    return 0;
}
```

- Extended keywords such as `pascal`, `inline`, `packed`, `interrupt`, `extended`, `__X`, `__Y`, `vector`, `pixel`, `bool` and others:

You typically use the compiler flag `-Xkeywords=`. For your Polyspace analysis, use `-compiler-parameter -Xkeywords=0xFFFFFFFF`

The following code will not compile with Polyspace unless you specify the above option:

```
packed(4) struct s2_t {
    char b;
    int i;
} s2;

packed(4,2) struct s3_t {
    char b;
} s3;

int pascal foo = 4;

int main(int argc, char** argv) {
    foo++;
    return 0;
}
```

Note that the Polyspace option only allows the code to be compiled. The analysis does not fully support the semantics behind the `packed` keyword.

Fix Polyspace Compilation Errors Related to Green Hills Compiler

If you choose `greenhills` for the option `Compiler (-compiler)`, you encounter this issue.

Issue

During Polyspace analysis, you see an error related to vector data types specific to Green Hills target `rh850`. For instance, you see an error related to identifier `__ev64_u16__`.

Cause

When compiling code using the Green Hills compiler with target `rh850`, to enable single instruction multiple data (SIMD) vector instructions, you specify two flags:

- `-rh850_simd`: You enable intrinsic functions that support SIMD vector instructions. The functions are defined in your compiler header files. These data types are available:
 - `__ev64_u16__`
 - `__ev64_s16__`
 - `__ev64_u32__`
 - `__ev64_s32__`
 - `__ev64_u64__`
 - `__ev64_s64__`
 - `__ev64_opaque__`
 - `__ev128_opaque__`
- `-rh850_fpsimd`: You enable intrinsic functions that support floating-point SIMD vector instructions. The functions are defined in your compiler header files. These data types are available:
 - `__ev128_f32__`
 - `__ev256_f32__`

The Polyspace analysis does not enable SIMD support by default. You must identify your compiler flags to Polyspace.

Solution

In your Polyspace analysis, use the command-line option `-compiler-parameter`. In the user interface, you can enter the command-line option in the `Other` field, under the **Advanced Settings** in the **Configuration** pane.

- `-rh850_simd`: For your Polyspace analysis, use
`-compiler-parameter -rh850_simd`
- `-rh850_fpsimd`: For your Polyspace analysis, use
`-compiler-parameter -rh850_fpsimd`

Note

- `__ev128_opaque__` is 16 bytes aligned in Polyspace.
 - `__ev256_f32__` is 32 bytes aligned in Polyspace.
-

Fix Polyspace Compilation Errors Related to TASKING Compiler

If you choose tasking for the option Compiler (-compiler), you can encounter this issue.

Issue

During Polyspace analysis, you see an error related to a Special Function Register data type.

Cause

When compiling with the TASKING compiler, you typically use the following compiler flags to specify where Special Function Register (SFR) data types are declared:

- `--cpu=xxx`: The compiler implicitly `#includes` the file `sfr/regxxx.sfr` in your source files. Once `#include`-ed, you can use Special Function Registers (SFR-s) declared in that `.sfr` file.
- `--alternative-sfr-file`: The compiler uses an alternative SFR file instead of the regular SFR file. You can use Special Function Registers (SFR-s) declared in that alternative SFR file.

If you specify the TASKING compiler for your Polyspace analysis, the analysis makes the following assumptions about these compiler flags:

- `--cpu=xxx`: The analysis chooses a specific value of `xxx`. If you use a different value with your TASKING compiler, you can encounter an error during Polyspace analysis.

The `xxx` value that the Polyspace analysis uses depends on your choice of Target processor type (-target):

- `tricore: tc1793b`
- `c166: xc167ci`
- `rh850: r7f701603`
- `arm: ARMv7M`
- `--alternative-sfr-file`: The analysis assumes that you do not use an alternative SFR file. If you use one, you can encounter an error.

Solution

Use the command-line option `-compiler-parameter` in your Polyspace analysis as follows. You use this command-line option to make Polyspace aware of your compiler flags. In the user interface, you can enter the command-line option in the field `Other`. You can enter the option multiple times.

- `--cpu=xxx`: For your Polyspace analysis, use

```
-compiler-parameter --cpu=xxx
```

Here, `xxx` is the value that you use when compiling with your compiler.

- `--alternative-sfr-file`: For your Polyspace analysis, use

```
-compiler-parameter --alternative-sfr-file
```

If you still encounter an error because Polyspace is not able to locate your `.asfr` file, explicitly `#include` your `.asfr` file in the preprocessed code using the option `Include` (-include).

Typically, the path to the file is *Tasking_C166_INSTALL_DIR*\include\sfr*regCPUNAME*.asfr. For instance, if your TASKING compiler is installed in C:\Program Files\Tasking\C166-VX_v4.0r1\ and you use the CPU-related flag -Cxc2287m_104f or --cpu=xc2287m_104f, the path is C:\Program Files\Tasking\C166-VX_v4.0r1\include\sfr\regxc2287m.asfr.

You can also encounter the same issue with alternative sfr files when you trace your build command. For more information, see “Requirements for Project Creation from Build Systems” on page 13-24.

Fix Polyspace Compilation Errors Related to Texas Instruments Compilers

Issue

The Texas Instruments compilers impose a nonstandard requirement on folder sequences in the include file search path, which is not directly supported by the Polyspace project creation mechanism from build systems (using the `polyspace-configure` command).

As a result, if you create a Polyspace project from a build that uses a Texas Instruments compiler, after starting an analysis on the project, you might see the warning:

```
warning: could not find include file "stddef.h"
| #include_next <stddef.h>
|
```

Possible Solutions

The Texas Instruments compilers impose a certain order of include folders in the include file search path. In particular, the compilers require the implicitly specified `libcxx` subfolder of the compiler include folder to *precede* explicit subfolders in the include file search path. When you create a Polyspace project or options file by tracing a build command, the project or options file contains an include folder sequence where the implicitly included `libcxx` subfolder *comes after* explicit subfolders. As a result, `include_next` lines in files in the `libcxx` subfolder, which only use later include folders in the search path for include file lookup, fail to find the included files.

To work around the problem:

- 1 In your Polyspace project or options file, locate the `-I` options that point to `libcxx` subfolders. They will appear in lines starting with `-options-for-sources`, for instance:

```
-options-for-sources sourcefile.c;-I some_explicitly_included_folder;-I
compiler_include_folder\libcxx;
```

- 2 For each such `-options-for-sources` line, switch the order of the `-I`-s so that the `libcxx` subfolder appears first, for instance:

```
-options-for-sources sourcefile.c;-I compiler_include_folder\libcxx;-I
some_explicitly_included_folder;
```

See Also

`polyspace-configure` | Texas Instruments Compiler (`-compiler ti`) | `-options-for-sources` | `-I`

Fix Errors from Use of Polyspace Header Files

Issue

When analyzing your C/C++ source code with Polyspace, if you do not provide the paths to your compiler headers, Polyspace uses its own version of the headers for the analysis.

In some cases, you might see compilation errors from these Polyspace headers. The error messages typically refer to one of the subfolders of *polyspaceroot*\polyspace\verifier\cxx\include. Here, *polyspaceroot* is the Polyspace installation folder, for instance, C:\Program Files\Polyspace\R2023a. Typically, the error message is related to a standard library function.

For instance, you might see an error on `std::is_empty` when analyzing this C++14 code:

```
#include <type_traits>

struct S final { };

bool f(void) {
    return std::is_empty<S>::value;
}
```

The error message states:

```
Error: a "final" class type cannot be used as a base class
```

And points to the path *polyspaceroot*\polyspace\verifier\cxx\include\include-libcxx\type_traits. This error happens because the implementation of `std::is_empty` in Polyspace header files in some cases do not allow their instantiations to use `final` classes.

Possible Solutions

Specify the folders containing *your compiler header files* for the Polyspace analysis.

If you create a Polyspace project or options file from your build command using `polyspace-configure`, the compiler header paths are automatically added to this project or options file. Otherwise, you have to explicitly add these paths:

- In the user interface, add the folders to your project.

For more information, see “Add Source Files for Analysis in Polyspace Desktop User Interface” on page 2-2.

- At the command line, use the flag `-I` with the `polyspace-bug-finder` or `polyspace-bug-finder-server` command.

For more information, see `-I`.

For compilation with GNU C on UNIX-based platforms, use `/usr/include`. On embedded compilers, the header files are typically in a subfolder of the compiler installation folder. Examples of include folders are given for some compilers.

- Wind River Diab: For instance, `/apps/WindRiver/Diab/5.9.4/diab/5.9.4.8/include/`.
- IAR Embedded Workbench: For instance, `C:\Program Files\IAR Systems\Embedded Workbench 7.5\arm\inc`.

- Microsoft Visual Studio: For instance, C:\Program Files\Microsoft Visual Studio 14.0\VC\include.

Consult your compiler documentation for the path to your compiler header files. Alternatively, see “Provide Standard Library Headers for Polyspace Analysis” on page 13-20.

See Also

Related Examples

- “Provide Standard Library Headers for Polyspace Analysis” on page 13-20

Fix Polyspace Compilation Errors About Namespace `std` Without Prefix

Issue

The Polyspace analysis stops with an error message such as:

```
error: the global scope has no "modfl"
```

The line highlighted in the error uses a function from the standard library without the `std::` prefix.

Cause

Some compilers allow using members of the standard library namespace without explicitly specifying the `std::` prefix. For such compilers, your code can contain lines like this:

```
using ::mblen;
```

where `mblen` is a member of the C++ standard library. Polyspace compilation considers the members as part of the global namespace and shows an error.

Solution

It is a good practice to qualify members of the standard library with the `std::` prefix. For instance, to use the `mblen` function in the preceding example, rewrite the line as:

```
using std::mblen;
```

To continue to retain the current code and work around the Polyspace error, use the analysis option `-using-std`. If you are running the analysis in the Polyspace user interface, enter the option in the **Other** field. See **Other**.

Fix Polyspace Compilation Warnings Related to Assertion or Memory Allocation Functions

Issue

Polyspace uses its own implementation of standard library functions for more efficient analysis. If you redefine a standard library function and provide the function body to Polyspace, the analysis uses your definition.

However, for certain standard library functions, Polyspace continues to use its own implementations, even if you redefine the function and provide the function body. The functions include `assert` and memory allocation functions such as `malloc`, `calloc` and `alloca`.

You see a warning message like the following:

```
Body of routine "malloc" was discarded.
```

Cause

These functions have special meaning for the Polyspace analysis, so you are not allowed to redefine them. For instance:

- The Polyspace implementation of the `malloc` function allows the software to check if memory allocated using `malloc` is freed later.
- The Polyspace implementation of `assert` is used internally to enhance analysis.

Solution

Unless you particularly want your own redefinitions to be used, ignore the warning. The analysis results are based on Polyspace implementations of the standard library function, which follow the original function specifications.

If you want your own redefinitions to be used and you are sure that your redefined function behaves the same as the original function, rename the functions. You can rename the function only for the purposes of analysis using the option `Preprocessor definitions (-D)`. For instance, to rename a function `malloc` to `my_malloc`, use `malloc=my_malloc` for the option argument.

Fix Polyspace Compilation Errors About In-Class Initialization (C++)

When a data member of a class is declared `static` in the class definition, it is a *static member* of the class. You must initialize static data members outside the class because they exist even when no instance of the class has been created.

```
class Test
{
public:

    static int m_number = 0;
};
```

Error message:

Error: a member with an in-class initializer must be const

Corrected code:

in file Test.h	in file Test.cpp
<pre>class Test { public: static int m_number; };</pre>	<pre>int Test::m_number = 0;</pre>

Update Eclipse Java Version for Polyspace Plug-in

In this section...
"Issue" on page 32-47
"Cause" on page 32-47
"Solution" on page 32-47

Issue

After installing the Polyspace plug-in for Eclipse, when you open the Eclipse or Eclipse-based IDE, you see this error message:

```
Java 7 required, but the current java version is 1.6.  
You must install Java 7 before using Polyspace plug in.
```

You might see this message even if you install Java 7 or higher.

Cause

Despite installing Java 7 or higher, the Eclipse or Eclipse-based IDE still uses an older version.

Solution

Make sure that the Eclipse or Eclipse-based IDE uses the compatible Java version.

- 1 Open the *executable_name.ini* file that occurs in the root of your Eclipse installation folder.

If you are running Eclipse, the file is *eclipse.ini*.

- 2 In the file, just before the line `-vmargs`, enter:

```
-vm  
java_install\bin\javaw.exe
```

Here, *java_install* is the Java installation folder.

For instance, your product installation comes with the required Java version for certain platforms. You can force the Eclipse or Eclipse-based IDE to use this version. In your *.ini* file, enter the following just before the line `-vmargs`:

```
-vm  
polyspaceroot\sys\java\jre\arch\jre\bin\javaw.exe
```

Here, *polyspaceroot* is your product installation folder, for instance, `C:\Program Files\Polyspace\R2019a\` and *arch* is `win32` or `win64` depending on the product platform. Note that `-vm` and the path to `javaw.exe` must be on separate lines.

Fix MATLAB Crashes Referring to Polyspace in matlabroot

Issue

In your Polyspace installation, you can find a MATLAB executable in the *polyspaceroot*\bin subfolder. The reason is that some functionalities of Polyspace use the MATLAB engine underneath. However, the MATLAB engine shipped with Polyspace is severely reduced and cannot be used by end-users even after license activation.

If you try to open `matlab.exe` from a Polyspace installation folder and try to execute MATLAB commands, MATLAB might crash during command execution. The crash log shows that you opened MATLAB from a Polyspace installation folder, such as `C:\Program Files\Polyspace\R2023a`.

Possible Solutions

Do not open MATLAB from a Polyspace installation by running an executable such as:

```
C:\Program Files\Polyspace\R2023a\bin\matlab.exe
```

Instead, open MATLAB from an actual MATLAB installation by running an executable such as:

```
C:\Program Files\MATLAB\R2023a\bin\matlab.exe
```

To see which MATLAB installation you are using, at the MATLAB command window, enter:

```
matlabroot
```

This command shows you the root of the MATLAB installation.

Note that you can run Polyspace from a MATLAB command line. But even for this usage, you must open MATLAB from your *MATLAB installation folder* and run some preliminary steps to integrate your MATLAB and Polyspace installation. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

See Also

Related Examples

- “Integrate Polyspace with MATLAB and Simulink” on page 5-2
- “Bug Finder Analysis with MATLAB Scripts”

Diagnose Why Coding Standard Violations Do Not Appear as Expected

Issue

You expect a coding rule violation on a line of code but the Polyspace Bug Finder analysis does not show the violation.

The default code analysis with Polyspace Bug Finder might not raise some potential defects. If an expected defect does not appear, you might need to enable additional checkers, and specify checker options that are appropriate for your analysis. For instance:

```
void foo(int* p, int i) {
    *p = i;
}
```

In this code example, the function `foo` dereferences a pointer with an unknown origination point. You might expect Bug Finder to flag the pointer as tainted. A default Bug Finder analysis does not look for tainted pointers. To flag pointers that originate outside the current analysis perimeter as tainted, enable the checker `TAINTED_PTR`. Then, limit your trust boundary by using the option `-consider-analysis-perimeter-as-trust-boundary`.

Possible Solutions

Check If Rule Checker Is Enabled

You might have enabled a reduced subset of coding rules that exclude the expected rule.

For instance, if you check for MISRA C: 2012 rules, the analysis enables only the `mandatory-required` subset by default. To check for other MISRA C: 2012 rules, specify the appropriate option.

Review the coding rules options that you use. See “Coding Standards & Code Metrics”.

Check If Source File Is Excluded from Analysis

By default, coding rule violations are suppressed from header files that are not in the same location as the source files. Some source files might be excluded from the coding rule analysis intentionally. See `Do not generate results for (-do-not-generate-results-for)`. If you are analyzing such a file, all coding rule violations in the file might be suppressed.

Check If Rule Violation Is in Macros

When a rule violation occurs in a macro definition, Polyspace shows the result only once in the macro definition. A violation is reported on an instance of a macro only when the rule violation occurs explicitly because of the parameters of the macro instance. If the macro definition occurs in a header file, it might be suppressed from the results. See also “Polyspace Results in Lines Containing Macros” on page 31-12. If you expect a coding rule violation on a macro:

- Check the macro definition for the violation.
- If the macro definition is in a header file, check if you are suppressing coding rule violations from header files. See `Do not generate results for (-do-not-generate-results-for)`.

On the **Source** pane, to see if a line contains a macro expansion, look for the  icon.

```
LOG(gdeb,var,"foo() function");
```

Check for Compilation Errors

If any source file in the analysis does not compile, coding rules checking remains incomplete. The coding rules checker results:

- Might not contain full results for files that did not compile.
- Might not contain full results for the files that did compile because some rules are checked only after compilation is complete.

Check for compilation errors. See “View Error Information When Analysis Stops” on page 32-4.

Check If Rule Checker Needs to Be Extended

Some constructs in your code might violate a coding rule in some specific contexts. A default Bug Finder analysis might not report these potential violations to reduce false positives. Detect these issues by extending the existing Bug Finder coding rule checkers. Modify the checker behavior by using these options:

- `-consider-analysis-perimeter-as-trust-boundary`: By default, Polyspace considers only the user inputs as tainted. By using this option, you consider any data that originates outside the current analysis perimeter as tainted. See “Sources of Tainting in a Polyspace Analysis” on page 18-61.
- `-detect-atomic-data-race`: By default, Polyspace assumes that certain operations are atomic and excludes them from data race checks. These operations might not be atomic in your environment. Use this option to extend the data race checkers to include the assumed atomic operations. See “Extend Data Race Checkers to Atomic Operations” on page 18-35.

For a list of options that extend or modify Bug Finder checkers, see “Modify Checker Behavior” on page 18-3.

Check If Analysis Needs Additional Information on Code

Polyspace might require additional information about your code to detect certain coding rule violations. For instance:

- Standard library math function checkers. By default, these checkers check for invalid use of standard library math functions. If you use a custom math library with domain or other constraints, a default Polyspace analysis cannot detect the violation of those constraints. If your custom library functions have similar domains and constraints to standard library math functions, you can extend these checkers to your custom library. See “Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries” on page 18-24.
- Concurrency checkers. By default, Polyspace Bug Finder automatically detects thread creation and protection mechanisms if you use routines from certain libraries. See “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 15-7. If you use a custom concurrency library that Polyspace does not support, to detect potential defects, do either of these:
 - If possible, map the custom library to a supported library. See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments” on page 18-30.
 - Configure the analysis manually. See “Configuring Polyspace Multitasking Analysis Manually” on page 15-17.

- Checkers that check for blocklisted keywords and functions. These checkers cannot flag the blocklisted keywords and functions unless you specify which ones are blocklisted. See “Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers” on page 18-21.

For a complete list of checkers that need additional information about your code, see “Modify Polyspace Interpretation of Code” on page 18-5.

Check If Rule Applies to Source Files

Some coding rules apply only to header files. Violations of these rules are not reported in nonheader source files. For instance, violations of rules such as MISRA C++:2008 Rule 3-1-1 or AUTOSAR C++14 Rule A3-1-1 are reported only in header files.

Check If Rule Violation Is in Uncalled static Function

Violations of some coding rules are not reported on uncalled `static` functions. To see all detect coding rule violations in a `static` function, call the function in your code.

Check If Violation Is in Unused Code

Bug Finder does not report coding rule violations on certain code constructs if they are unused. For instance:

- Templates: Bug Finder does not report violations on uninstantiated templates.
- Macros: Bug Finder does not report violations on unused macros.
- Typedefs: Bug Finder does not report violations on unused typedefs.

To check these code constructs for coding rule violations, invoke them at least once in your code.

See Also

Do not generate results for `(-do-not-generate-results-for) | -code-behavior-specifications | -consider-analysis-perimeter-as-trust-boundary | -detect-atomic-data-race`

More About

- “Modify Default Behavior of Bug Finder Checkers” on page 18-3
- “Sources of Tainting in a Polyspace Analysis” on page 18-61
- “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 15-7
- “Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers” on page 18-21
- “Bug Finder Analysis Assumptions”

Check Why a Bug Finder Defect Does Not Appear as Expected

Issue

The default code analysis with Polyspace Bug Finder might not raise some potential defects. If an expected defect does not appear, you might need to enable additional checkers, and specify checkers options that are appropriate for your analysis. For instance:

```
void foo(int* p, int i) {  
    *p = i;  
}
```

In this code example, the function `foo` dereferences a pointer with an unknown origination point. You might expect Bug Finder to flag the pointer as tainted. A default Bug Finder analysis does not look for tainted pointers. To flag pointers that originate outside the current analysis perimeter as tainted, enable the checker `TAINTED_PTR`. Then, limit your trust boundary by using the option `-consider-analysis-perimeter-as-trust-boundary`.

Possible Solutions

Check If Defect Checker Is Enabled

The default Bug Finder analysis enables a subset of the defect checkers. See “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 18-65. To enable for other Bug Finder defects:

- Polyspace desktop user interface: Select the checkers that you want to enable in the **Bug Finder Analysis** node of the **Configuration** pane.
- Polyspace as You Code IDE plugins: Select the checkers that you want to enable in the Checkers Selection window. See “Setting Checkers in Polyspace as You Code”.
- Command Line Interface: Open the Checkers Selection window by using the command `polyspace-checkers-selection`. Select the checkers that you want to enable, and then save your selection in an XML file. When running a Bug Finder analysis, use this file as the input to the command `-checkers-activation-file`. Instead of specifying a file, you can also enumerate the checkers explicitly or use a predefined subset using the option `Find defects (-checkers)`.

When checking for specific issues, the best practice is to enable specific checkers corresponding to the issues.

Check If Defect Checker Needs to Be Extended

Some constructs in your code might cause issues in some specific contexts. A default Bug Finder analysis might not flag these potential defects to reduce false positives. Detect these issues by extending the existing Bug Finder defect checkers. Modify the checker behavior by using these options:

- `-consider-analysis-perimeter-as-trust-boundary`: By default, Polyspace considers only the user inputs as tainted. By using this option, you consider any data that originates outside the current analysis perimeter as tainted. See “Sources of Tainting in a Polyspace Analysis” on page 18-61.
- `-detect-atomic-data-race`: By default, Polyspace assumes that certain operations are atomic and excludes them from data race checks. These operations might not be atomic in your

environment. Use this option to extend the data race checkers to include the assumed atomic operations. See “Extend Data Race Checkers to Atomic Operations” on page 18-35.

For a complete list of options that extend or modify Bug Finder checkers, see “Modify Checker Behavior” on page 18-3.

Check If Analysis Needs Additional Information on Code

Some checkers might require additional information about your code in order to detect certain defects. For instance:

- Standard library math function checkers. By default, these checkers check for invalid use of standard library math functions. If you use a custom math library with domain or other constraints, a default Polyspace analysis cannot detect the violation of those constraints. If your custom library functions have similar domains and constraints to standard library math functions, you can extend these checkers to your custom library. See “Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries” on page 18-24.
- Concurrency checkers. By default, Polyspace Bug Finder automatically detects thread creation and protection mechanisms if you use routines from certain libraries. See “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 15-7. If you use a custom concurrency library that Polyspace does not support, to detect potential defects, do either of these:
 - If possible, map the custom library to a supported library. See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments” on page 18-30.
 - Configure the analysis manually. See “Configuring Polyspace Multitasking Analysis Manually” on page 15-17.
- Checkers that check for blocklisted keywords and functions. These checkers cannot flag the blocklisted keywords and functions unless you specify which ones are blocklisted. See “Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers” on page 18-21.

For a complete list of checkers that need additional information about your code, see “Modify Polyspace Interpretation of Code” on page 18-5.

Check Bug Finder Assumptions

To minimize results that can be perceived as false positives, Bug Finder makes certain assumptions when running a Bug Finder analysis. For instance:

- By default, Polyspace does not raise a defect that is caused by a specific value of an unknown input. See “Inputs in Polyspace Bug Finder”.
- Polyspace ignores the initialization value of local `volatile` variables. The initialization value of global `volatile` objects depend on their `const`-ness and their **Init Mode**. See “Volatile Variables in Polyspace Bug Finder”.

For a list of these assumptions, see “Bug Finder Analysis Assumptions”.

These assumptions might prevent Bug Finder from detecting some potential defects. In these cases, run Code Prover for a more exhaustive analysis. For instance, consider this code:

```
int foo(void){
    volatile var=0;
    return 1/var; // Potential defect
```

```
}
```

The statement `return 1/var` is a potential divide-by-zero error. Because of Bug Finder assumptions, this defect is not detected. A Code Prover analysis flags this issue as an orange check.

Check For Uninstantiated Template

If your code has uninstantiated class template, a Polyspace Bug Finder defect might not be reported on the class.

The objects in standard template library (STL) are class templates. If you do not see an expected defect on an STL class, instantiate an object of the class.

See Also

`-code-behavior-specifications` | `-consider-analysis-perimeter-as-trust-boundary` | `-detect-atomic-data-race` | Effective boolean types (`-boolean-types`)

More About

- “Modify Default Behavior of Bug Finder Checkers” on page 18-3
- “Sources of Tainting in a Polyspace Analysis” on page 18-61
- “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 15-7
- “Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers” on page 18-21
- “Bug Finder Analysis Assumptions”

Fix Insufficient Memory Errors During Polyspace Report Generation

Issue

When generating reports from Polyspace results containing a very large number of defects or coding rule violations, you might encounter insufficient memory errors.

The error message can look like this message:

```

....
Exporting views...
Initializing...
Polyspace Report Generator
Generating Report
.....
  Converting report
Opening log file: C:\Users\auser\AppData\Local\Temp\java.log.7512
Document conversion failed
.....
Java exception occurred:
java.lang.OutOfMemoryError: Java heap space

```

Possible Solutions

To resolve the issue, you can try increasing the available heap memory or reporting the results over multiple reports instead of in a single report.

Increase Java Heap Size

If the error occurs during report generation, try increasing the Java heap size. The default heap size in a 64-bit architecture is 1024 MB.

To increase the size:

- 1 Navigate to *polyspaceroot*\polyspace\bin*architecture*. Where:
 - *polyspaceroot* is the installation folder.
 - *architecture* is your computer architecture, for instance, win32, win64, etc.
- 2 Change the default heap size that is specified in the file, *java.opts*. For example, to increase the heap size to 2 GB, replace *1024m* with *2048m*.
- 3 If you do not have write permission for the file, copy the file to another location. After you have made your changes, copy the file back to *polyspaceroot*\polyspace\bin*architecture*.

Report Polyspace Results in Multiple Reports

Instead of reporting all results from a result set (*.pscp* or *.psbf* file) in a single report, you can generate multiple reports, each containing a smaller subset of results.

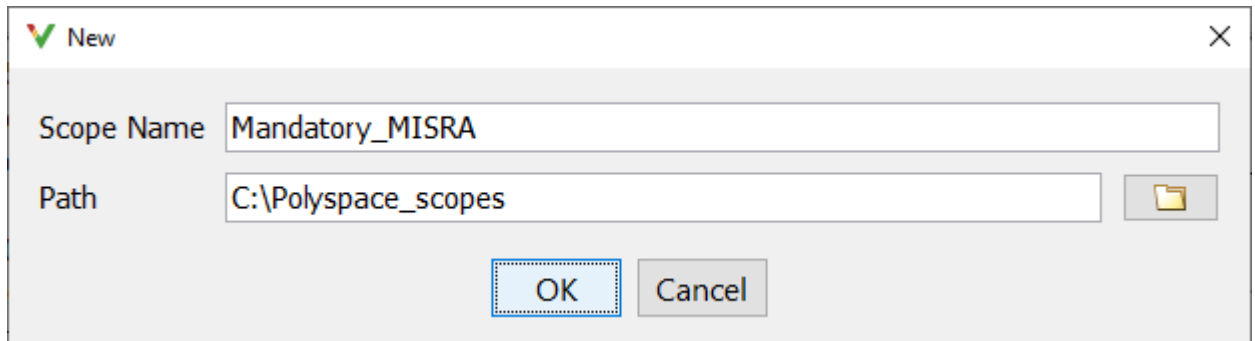
The simplest strategy can be to report results of a certain type in a single report. For instance, from a results set containing MISRA C:2012 rule violations, you can generate two reports, one for the mandatory and required rules, and another for the remaining rules.

Apply Review Scopes and Generate Filtered Report

You can create named sets of filters called review scopes in the user interface of the Polyspace desktop products. When generating a report, you can apply a review scope that filters the results before storing them in the report.

To create a review scope:

- 1 In the Polyspace user interface, select **Tools > Preferences**.
- 2 On the **Review Scope** tab, select **New**. Enter a review scope name and a location to save.



The review scope name can be used later when filtering the results.

- 3 Select the checkers that you want to be included in the review scope.

You can generate a filtered report from a Polyspace results set by applying a review scope before report generation. The report contains results of only those checkers that are included in the review scope. Note that even though you create the review scope in the user interface, you can apply the scope for report generation in the user interface or at the command line.

To generate a filtered report in the Polyspace user interface:

- 1 In the dropdown at the top of the **Results List** pane, instead of **All results**, select your new review scope. The list of results is narrowed down to results of only those checkers that are included in the review scope.
- 2 Generate a report or export results using the **Reporting** menu.
 - To generate reports, select **Run Report**. In the Run Report dialog, select **Only include currently displayed results** before starting report generation.
 - To export results select **Export > Export Currently Displayed Results**.

To generate a filtered report at the command line, use the option `-wysiwyg scopeName` with the command `polyspace-report-generator`. Here, `scopeName` is the review scope name that you used when saving the scope from the Polyspace user interface. For instance:

```
polyspace-report-generator
  -template templateName
  -results-dir resultsFolder
  -format HTML -wysiwyg scopeName
```


Here:

- *templateName* is the path to a report template, such as *polyspaceroot\toolbox\polyspace\psrptgen\templates\Developer.rpt*, where *polyspaceroot* is the Polyspace installation folder such as *C:\Program Files\Polyspace\R2023a*.
- *resultsFolder* is the folder containing Polyspace results.

Note that the scope names are stored with your user preferences. Therefore, you cannot use the scopes names for report generation with another Polyspace installation, for instance, from a different release. For more information on user preferences, see “Storage of Polyspace User Interface Customizations” on page 2-24.

Apply Column-Based Results List Filters and Generate Filtered Report

You can also filter results in the Polyspace user interface using column-based filters and create a filtered report from the currently displayed results:

- 1 Click the  icon on column headers of the **Results List** pane to see the available filters. Apply the filters that you want.
- 2 Generate a report or export results using the **Reporting** menu.
 - To generate reports, select **Run Report**. In the Run Report dialog, select **Only include currently displayed results** before starting report generation.
 - To export results select **Export > Export Currently Displayed Results**.

The generated report states which columns were used to filter results. For instance, if you use the **Status** column to suppress all results with status *Justified*, the generated report contains this line:

```
Columns with active filters:
  Status
```

All results with status *Justified* do not appear in the report.

Your choice of column-based filters are stored in the file *ui_settings.prf* in the *.settings* subfolder of the results folder. Therefore, if you generate a report from these results at the command line and use the option `-wysiwyg "All results"` (or with a scope name as shown in previous section), your choice of column-based filters apply to the generated report. You can even move the file *ui_settings.prf* to *.settings* subfolders of other results folders to generate filtered reports from those other results.

See Also

`polyspace-report-generator`

More About

- “Generate Reports from Polyspace Results” on page 24-2

Fix Errors or Slow Polyspace Runs from Disk Defragmentation and Anti-virus Software

Issue

In some cases, anti-virus software checks can noticeably slow down a Polyspace analysis. This reduction occurs because the software checks the temporary files produced by the Polyspace analysis.

You see noticeably slow analysis for a simple project or the analysis stops with an error message like the following:

```
Some stats on aliases use:
  Number of alias writes:      22968
  Number of must-alias writes: 3090
  Number of alias reads:      0
  Number of invisibles:       949
Stats about alias writes:
  biggest sets of alias writes: foo1:a (733), foo2:x (728), foo1:b (728)
  procedures that write the biggest sets of aliases: foo1 (2679), foo2 (2266),
                                                    foo3 (1288)
**** C to intermediate language translation - 17 (P_PT) took 44real, 44u + 0s (1.4gc)
exception SysErr(OS.SysErr(name="Directory not empty", syserror=notempty)) raised.
unhandled exception: SysErr: No such file or directory [noent]
```

```
-----
---
--- Verifier has encountered an internal error.      ---
--- Please contact your technical support.          ---
---
-----
```

Possible Cause

A disk defragmentation tool or anti-virus software is running on your machine.

After starting an analysis, check the processes running and see if an anti-virus process is causing large amount of CPU usage (and possibly memory usage).

Solution

Try:

- Stopping the disk defragmentation tool.
- Deactivating the anti-virus software. Or, configuring exception rules for the anti-virus software to allow Polyspace to run without a failure.

For instance, you can try the following:

- Configure the anti-virus software to allow the Polyspace executables.

For instance, in Windows, with the anti-virus software Windows Defender, you can add an exclusion for the Polyspace installation folder C:\Program Files\Polyspace\R2019a, in particular, the .exe files in the subfolder polyspace\bin and the .exe files starting with polyspace in the subfolder bin\win64 (for instance, polyspace-internal-connector.exe).

- Configure the anti-virus software to exclude your temporary folder, for example, C:\Temp, from the checking process.

See “Storage of Temporary Files During Polyspace Analysis” on page 3-6.

Fix SQLite I/O Errors on Running Polyspace

Issue

Polyspace uses an SQLite database for storing analysis results. SQLite databases might show problems when shared on network file systems such as NFS (Network File System), CIFS (Common Internet File System) or SMB (Server Message Block protocol), and the like.

If you save your analysis results on network file systems, you might see errors like this:

```
exception SQLError(SQLite.SQLError(code=10, disk I/O error (errcode=10 extended errcode=1034)))  
raised.
```

The errors indicate that the database disk image is malformed and the results are possibly corrupted.

Possible Solutions

Check the folder where you save Polyspace results. For instance, if you run Polyspace at the command line, check the argument of the option `-results-dir`.

If the folder is a network folder that uses file systems such as NFS, use a local folder instead.

Fix Polyspace Errors Related to Temporary Files

Polyspace produces some temporary files during analysis. The following issues are related to storage of temporary files.

No Access Rights

When running verification, you get an error message that Polyspace could not create a folder for writing temporary files. For instance, the error message can be as follows:

```
Unable to create folder "C:\Temp\Polyspace\foldername
```

Cause

Polyspace produces some temporary files during analysis. If you do not have write permissions for the folder used to store the files, you can encounter the error.

Solution

There are two possible solutions to this error:

- Change the permissions of your temporary folder so you have full read and write privileges.

To learn how Polyspace determines the temporary folder location, see “Storage of Temporary Files During Polyspace Analysis” on page 3-6.

- Use the option `-tmp-dir-in-results-dir`. Instead of the standard temporary folder, Polyspace uses a subfolder of the results folder.

No Space Left on Device

When running verification, you get an error message that there is no space on a device.

Cause

If you do not have sufficient space on for the folder used to store the files, you can encounter the error.

Solution

There are two possible solutions to this error:

- Change the temporary folder to a drive that has enough disk space.

To learn how Polyspace determines the temporary folder location, see “Storage of Temporary Files During Polyspace Analysis” on page 3-6.

- Use the option `-tmp-dir-in-results-dir`. Instead of the standard temporary folder, Polyspace uses a subfolder of the results folder.

Cannot Open Temporary File

When running verification, you get an error message that Polyspace could not open a temporary file.

Cause

You defined the path for storing temporary files by using the environment variable `RTE_TMP_DIR`. You either used a relative path for the temporary folder, the folder does not exist or you do not have access rights to the folder.

Solution

There are two possible solutions to this error:

- Instead of defining a temporary folder specific to Polyspace through `RTE_TMP_DIR`, use a standard temporary folder.

To learn how Polyspace determines the temporary folder location, see “Storage of Temporary Files During Polyspace Analysis” on page 3-6.

- If you continue to use `RTE_TMP_DIR`, make sure you specify an absolute path to an existing folder and you have access rights to the folder.

Fix Errors Applying Custom Annotation Format for Polyspace Results

Issue

When you use the option `-xml-annotations-description` to apply custom annotations to your Polyspace results, some custom annotations are not applied and you see warnings in the console output or the desktop interface **Output Summary**.

Possible Solutions

Custom Annotation Not Found in Mapping

If you define a custom annotation syntax but you do not map it to the Polyspace annotation syntax, Polyspace detects the custom annotation but does not apply it to the analysis results. You see a warning similar to this warning in the console output or the Polyspace desktop interface **Output Summary**.

```
Verifying sources ...
Verifying zero_div.c (1/1)
Warning: rule :50 from exampleCustomAnnotation not found in the mapping (XML file).
        Skipping the annotation
```

Solution

Check the `<Mapping/>` section of the XML file that you pass to the `-xml-annotations-description` option. If the rule listed in the warning is not mapped to a Polyspace rule, add the appropriate entry to map the rule. For instance, to map rule 50 from the preceding warning to Polyspace coding rule **MISRA C: 2012 Rule 8.4**, add this entry in the `<Mapping/>` section:

```
<Result_Name_Mapping Rule_Identifier="50" Family="MISRA-C3" Result_Name="8.4"/>
```

Polyspace Annotations Do Not Apply to Current Code

If you define a custom annotation syntax and you map it to the Polyspace annotation syntax, Polyspace might not apply some custom annotations to your source code. You see a warning similar to this warning in the console output or the Polyspace desktop interface **Output Summary**.

```
Warning: These Polyspace annotations do not apply to the current code:
|         In file D:\Polyspace\Examples\zero_div.c line 7, annotation MISRA-C3:8.4 with text
| Justified by annotation in source"
|         In file D:\Polyspace\Examples\zero_div.c line 20, annotation MISRA-C3:8.4 with text
| Justified by annotation in source"
|         Possible reasons:
|         - Issue not detected with selected configuration options.
|         - Issue is fixed.
|         - Annotation syntax is incorrect
```

Solution

Check for these possible causes:

- The issue that the annotation addresses has been fixed in the source code. Polyspace detects the custom annotation but ignores it.
- The issue that the annotation addresses was not detected by Polyspace with the analysis options that you specified. For example, if the custom annotation addresses a MISRA C: 2012 coding standard violation but Polyspace did not check for violations of this coding standard because option `Check MISRA C:2012 (-misra3)` is not specified.

- The issue that the annotation addresses was detected but Polyspace could not match the custom annotation to a corresponding Polyspace annotation. This indicates a syntax error in the `<Mapping/>` section of the XML file that you pass to the `-xml-annotations-description` option.

See Also

`-xml-annotations-description`

Related Examples

- “Define Custom Annotation Format” on page 30-30

Fix Issues When when Integrating Polyspace with MATLAB and Simulink

Issue

Before using Polyspace from MATLAB and Simulink, perform a one-time setup to integrate the two products. For details, see “Integrate Polyspace with MATLAB and Simulink” on page 5-2. When performing the integration steps, if you do not have administrator privileges in MATLAB or your Polyspace installation is nonstandard, you might run into some issues. For instance, the `polyspacesetup()` command might fail to complete.

Possible Solutions

Check if Polyspace installation is nonstandard

By default, Polyspace is installed on the path `C:\Program Files\Polyspace\R2023a`. When you use this command:

```
polyspacesetup('install');
```

Polyspace assumes that the installation folder is the default folder. If you install Polyspace in a different folder, the preceding command fails. For more information about the default installation folder, see “Product Installation”.

To resolve this issue, specify the installation path in the `polyspacesetup` command. For instance, at the command line, enter:

```
polyspacesetup('install', 'polyspaceFolder', Folder)
```

where *Folder* is the Polyspace installation folder.

Check if MATLAB instance has administrator privilege

Executing the command `polyspacesetup` requires administrator privilege. If you do not open MATLAB by using administrator privilege, the command exits with an error.

To resolve this issue, restart MATLAB by using administrator privileges. For instance, in Windows, to open MATLAB with administrator privilege, right-click the MATLAB executable and select **Run as administrator**. In some operating systems, you might need to use an administrator account.

Check If polyspacesetup expects user Input

By default, the `polyspacesetup` command is interactive and expects user input during the integration process. When performing a noninteractive integration, make sure the process is not stuck waiting for your input. To perform a noninteractive installation, at the MATLAB command prompt, enter:

```
polyspacesetup('install', 'polyspaceFolder', Folder, 'silent', true);
```

where *Folder* is the installation path.

Check if Polyspace version is supported

Polyspace integrates completely with MATLAB or Simulink from the same release. If your Polyspace and MATLAB are from different releases, you might not be able to perform a complete integration. See “Polyspace Support of MATLAB and Simulink from Different Releases” on page 6-68.

Depending on your versions of Polyspace and MATLAB, you might be able to partially integrate these products. See “MATLAB Release Earlier Than Polyspace” on page 5-3.

See Also

`polyspacesetup`

More About

- “Polyspace Support of MATLAB and Simulink from Different Releases” on page 6-68
- “Bug Finder Analysis with MATLAB Scripts”
- “Bug Finder Analysis in Simulink”
- “Bug Finder Analysis in MATLAB Coder”
- “Integrate Polyspace with MATLAB and Simulink” on page 5-2

Check Why Polyspace Functions are Unavailable in MATLAB

Issue

To use Polyspace directly from MATLAB or Simulink, you must include the folders containing the Polyspace functions on the MATLAB search path. If these locations are absent or deleted from the search path, the Polyspace functions become unavailable in the MATLAB Command Window.

Possible Solution

Possible Solution: Check if you integrated Polyspace with MATLAB and Simulink

The Polyspace functions are included with a Polyspace installation and their locations are unknown to MATLAB. After installing MATLAB and Polyspace, you cannot use these functions unless you add their locations to the MATLAB search path.

To use the Polyspace functions from MATLAB and Simulink, add their locations to the MATLAB search path by calling the function `polyspacesetup`. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

Possible Solution: Check if Polyspace is supported after a MATLAB update

Updating MATLAB restores the default MATLAB search path and removes any path to the Polyspace installation that was previously added. Because the Polyspace function paths are removed, the integration between Polyspace and MATLAB breaks after a MATLAB update. You must re-establish the integration.

After an update, repeat the steps to integrate Polyspace and MATLAB. If you updated MATLAB to a later release without updating Polyspace, the Polyspace support of MATLAB might be different from before. See “Polyspace Support of MATLAB and Simulink from Different Releases” on page 6-68.

Depending on your versions of Polyspace and MATLAB, integrate these products completely or partially. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

See Also

`polyspacesetup`

More About

- “Polyspace Support of MATLAB and Simulink from Different Releases” on page 6-68
- “Bug Finder Analysis with MATLAB Scripts”
- “Bug Finder Analysis in Simulink”
- “Bug Finder Analysis in MATLAB Coder”
- “Integrate Polyspace with MATLAB and Simulink” on page 5-2

Troubleshoot Java Incompatibility in Polyspace Plugin for Eclipse

Issue

Using the Polyspace desktop plugin for Eclipse and Eclipse-based IDEs require a Java version between 7 and 15. If your Java version is outside this range, after installing the plugin, you might see one of these error messages:

- Java 7 required, but the current java version is 1.6.
You must install Java 7 before using Polyspace plug in.
- Java version 16 is not supported by the Polyspace plugin

These messages indicate that you have an incompatible version of Java.

Possible Solutions

Check If Eclipse Uses Correct Java Installation

The Polyspace plugin for Eclipse and Eclipse-based IDEs requires a Java version between 7 and 15. If you have installed a compatible Java version, check if your Eclipse IDE is using that Java version:

- 1 In the Eclipse IDE, click **Help > About Eclipse IDE > Installation Details**.
- 2 In the Installation Detail window, on the **Configuration** tab, locate the line `java.version`. This line shows the Java version that is used by the IDE.

If the Java version is incompatible:

- 1 Install a Java version between 7 and 15, say, in the folder *java_install*. Polyspace comes with a compatible Java version in certain platforms. You might prefer to use this version of Java.
- 2 Open the *executable_name.ini* file from the Eclipse installation folder.

For the core Eclipse IDE, the file is `eclipse.ini`. For other Eclipse-based IDEs, the file name might be different.

- 3 In the file, before the line `-vmargs`, enter:

```
-vm
java_install\bin\javaw.exe
```

Here, *java_install* is the installation folder of the compatible Java version.

If you prefer to use the Java version Polyspace provides, then enter:

```
-vm
polyspaceroot\sys\java\jre\arch\jre\bin\javaw.exe
```

Here, *polyspaceroot* is your product installation folder, for instance, `C:\Program Files\Polyspace\R2019a\` and *arch* is `win32` or `win64` depending on the product platform. Note that `-vm` and the path to `javaw.exe` must be on separate lines.

Switch to Polyspace as You Code

If installing a compatible version of Java is not feasible, consider switching to the Polyspace as You Code plugin for Eclipse. You require a Polyspace Access license to install this plugin. For more information, see “Install Polyspace as You Code Plugin in Eclipse”.

Polyspace as You Code performs a Polyspace Bug Finder analysis on the source file currently open in your IDE. The plugin does not support checking for run-time errors and calculating stack usage by using Polyspace Code Prover. See also:

- “Analysis Scope of Polyspace as You Code” on page 11-75.
- “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78.

See Also

More About

- “Configure Polyspace as You Code Plugin in Eclipse” on page 11-19
- “Install Polyspace Desktop Plugin for Eclipse”
- “Install Polyspace as You Code Plugin in Eclipse”
- “Checkers Deactivated in Polyspace as You Code Analysis” on page 11-78

Troubleshooting Polyspace Access

Polyspace Access ETL and Web Server services do not start

Issue

You start the Polyspace Access services but after a moment, the **ETL** and **Web Server** services stop. You might see a HTTP 403 error message in your web browser when you try to connect to Polyspace Access.

Possible Cause: Hyper-V Network Configuration Cannot Resolve Local Host Names

On Windows, if you installed Polyspace Access inside a virtual machine (VM), that VM is managed by Hyper-V. Depending on your network configuration, Hyper-V might not resolve local host names. The **Polyspace Access ETL** and **Polyspace Access Web Server** services cannot connect to the host that you specify with these host names.

To test whether Hyper-V can resolve host name `myHostname` on a machine that is connected to the Internet, at the command line, enter:

```
docker run --rm -it alpine ping myHostname
```

If Hyper-V cannot resolve the host name, you get an error message.

Solution

Stop and restart the `admin-docker-agent` binary without using the `--hostname` option.

- If you are on a trusted network or you do not want to use the HTTPS protocol:
 - 1 At the command-line, enter:

```
docker stop admin  
  
admin-docker-agent --restart-gateway
```
 - 2 In the **Cluster Admin** web interface, click **Restart Apps**.
- If you want to use the HTTPS protocol, generate certificates with a subject alternative name (SAN) that includes the IP address of the cluster operator node on which the services are running.
 - 1 Copy this configuration file to a text editor and save it on your machine as `openssl.cnf`.

Configuration file

```
[ req ]  
req_extensions = v3_req  
distinguished_name = req_distinguished_name  
prompt = no  
  
[ req_distinguished_name ]  
countryName = US  
stateOrProvinceName = yourState  
organizationName = myCompany  
organizationalUnitName = myOrganization  
emailAddress = user@email.com  
commonName = hostName
```

```
[ v3_req ]
basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
subjectAltName = @alt_names

[ alt_names ]
DNS.1 = hostName
DNS.2 = fullyQualifiedDomainName
IP.1 = nodeIPAddress
```

hostName is the host name of the server that hosts Polyspace Access.
fullyQualifiedDomainName is the corresponding fully qualified domain name.
nodeIPAddress is the IP address of the node on which you run the `admin-docker-agent` binary.

You do not need to edit the value of the other fields in the `[req_distinguished_name]` section of `openssl.cnf`. Updating the value of these fields does not affect the configuration.

- 2 In the **Cluster Dashboard**, click **Configure Nodes**. The IP address listed in the **Hostname** field corresponds to *nodeIPAddress* in the `openssl.cnf` file. If there is more than one node listed, add an additional line in the `[alt_names]` section of `openssl.cnf` for each IP address. For example:

```
[ alt_names ]
DNS.1 = hostName
DNS.2 = fullyQualifiedDomainName
IP.1 = nodeIPAddress
IP.2 = additionalNodeIPAddress
```

- 3 Generate a certificate signing request (CSR) by using your `openssl.cnf` configuration file. At the command line, enter:

```
openssl req -new -out myRequest.csr -newkey rsa:4096 \
-keyout myKey.key -nodes -config openssl.cnf
```

The command outputs a private key file `myKey.key` and the file `myRequest.csr`.

- 4 To generate a signed certificate:
 - If you use your organization's certificate authority, submit `myRequest.csr` to the certificate authority. The certificate authority uses the file to generate a signed server certificate. For instance, `server_cert.cer`.
 - If you use self-signed certificates, at the command line, enter:

```
openssl x509 -req -days 365 -in myRequest.csr -signkey myKey.key \
-out self-cert.pem -extensions v3_req -extfile openssl.cnf
```

The command outputs self-signed certificate `self-cert.pem`.

- 5 Stop and restart the `admin-docker-agent` binary with this command:

Windows PowerShell	<code>./admin-docker-agent --restart-gateway \ --ssl-cert-file certFile1 \ --ssl-key-file keyFile \ --ssl-ca-file trustedStoreFile</code>
Linux	<code>./admin-docker-agent --restart-gateway \ --ssl-cert-file certFile1 \ --ssl-key-file keyFile \ --ssl-ca-file trustedStoreFile</code>

certFile1 is the full path of the file you obtained in step 4. *keyFile* is the file you generated in step 3. *trustedStoreFile* is the file you generated in step 4 if you used self-signed certificates. Otherwise, it is the trust store file you use to configure HTTPS. See “Choose Between HTTP and HTTPS Configuration for Polyspace Access” Save your changes.

- 6** In the **Cluster Admin** web interface, click **Restart Apps**.

Contact Technical Support About Polyspace Access Issues

If you need support from MathWorks for Polyspace Access issues, go to this page and create a service request. You need a MathWorks login and password to create a service request.


Before you contact MathWorks, gather this information.

- **Operating system**

To see information about the operating system of the machine where you install Polyspace access, at the command line, enter:


Windows	<code>systeminfo findstr /C:OS</code>
Linux	<code>uname -a</code>

- **Polyspace Access version**

Log into Polyspace Access, then at the top of the window click  > **About Polyspace**. If Polyspace Access is not yet installed or you cannot log into Polyspace Access, at the command line, navigate to the folder where you unzipped the Polyspace Access installation image, and enter:

Windows	<code>type VERSION</code>
Linux	<code>cat VERSION</code>

- **License number**

Log into Polyspace Access, then at the top of the window click  > **About Polyspace**. If Polyspace Access is not yet installed or you cannot log into Polyspace Access, contact your license administrator to obtain your license number.

- **Polyspace Access service logs**

To generate logs for the different Polyspace Access services, at the command line, enter:

```
docker logs -t polyspace-access-web-server-0-main >> access-web-server.log 2>&1
docker logs -t polyspace-access-etl-0-main >> access-etl.log 2>&1
docker logs -t polyspace-access-db-0-main >> access-db.log 2>&1
docker logs -t polyspace-access-download-0-main >> download-service.log 2>&1
docker logs -t issuetracker-server-0-main >> issuetracker-server.log 2>&1
docker logs -t issuetracker-ui-0-main >> issuetracker-ui.log 2>&1
docker logs -t usermanager-server-0-main >> usermanager-server.log 2>&1
docker logs -t admin >> admin.log 2>&1
docker logs -t gateway >> gateway.log 2>&1
docker logs -t usermanager-ui-0-main >> usermanager-ui.log 2>&1
docker logs -t usermanager-db-0-main >> usermanager-db.log 2>&1
docker logs -t polyspace-access >> polyspace-access.log 2>&1
docker logs -t issuetracker >> issuetracker.log 2>&1
docker logs -t usermanager >> usermanager.log 2>&1
```

Attach the log files to your service request. The commands to generate these logs are the same for Windows and Linux.

Note If you run Polyspace Access version R2021b or earlier, the docker container names might be different. To view the names of currently running containers, use command `docker ps --format '{{.Names}}'`.

- **Polyspace Access web interface log**

To generate a log for the Polyspace Access web interface, log into Polyspace Access. In the left pane, click **SUPPORT REPORT** then **Get support info**. Attach the generated support report file to your service request.