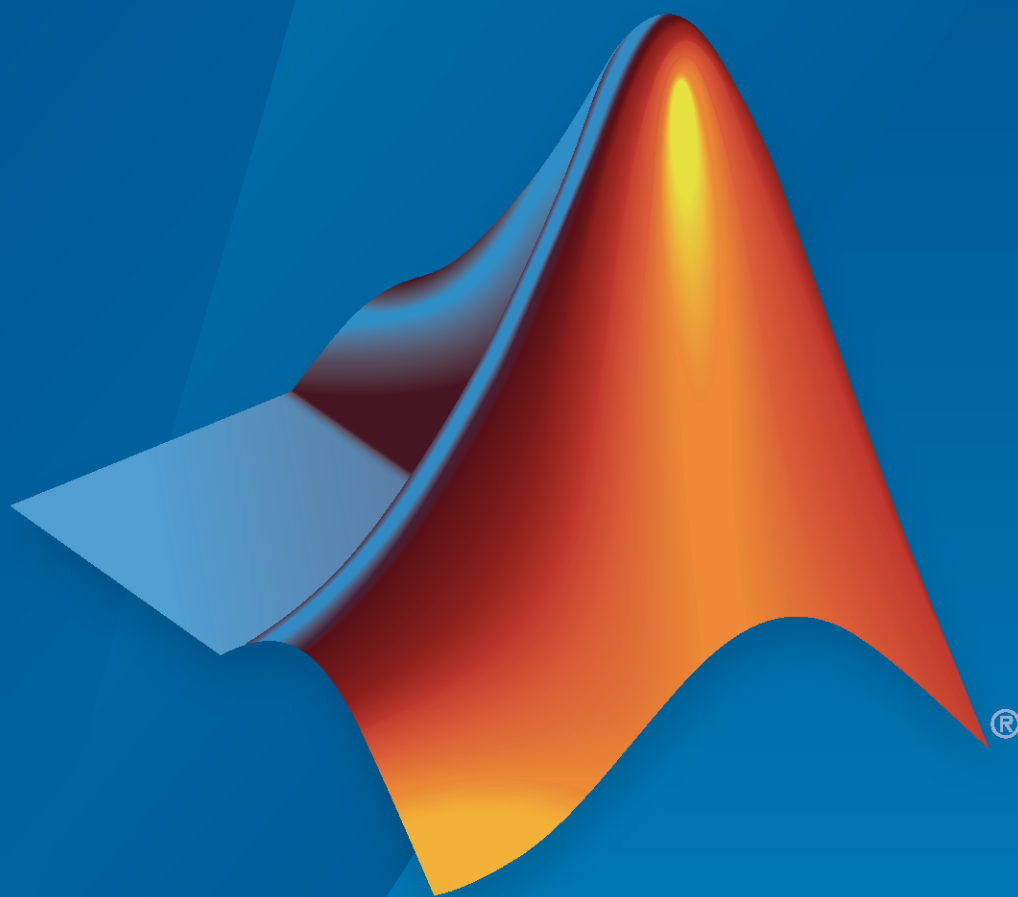


Polyspace® Bug Finder™

User's Guide



R2021a

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 2013–2021 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)

1

Run Polyspace Analysis on Desktop

Add Source Files for Analysis in Polyspace User Interface	1-2
Add Sources from Build Command	1-2
Add Sources Manually	1-4
Run Polyspace Analysis on Desktop	1-7
Arrange Layout of Windows for Project Setup	1-7
Set Product and Result Location	1-8
Start and Monitor Analysis	1-9
Fix Compilation Errors	1-9
Open Results	1-10
Project and Results Folder Contents	1-11
File Organization	1-11
Files in the Results Folder	1-11
Storage of Temporary Files	1-13
Create Project Using Visual Studio Information	1-14
Create Project Using Configuration Template	1-16
Why Use Templates	1-16
Use Predefined Template	1-16
Create Your Own Template	1-16
Update Polyspace Project	1-19
Change Folder Path	1-19
Refresh Source List	1-20
Refresh Project Created from Build Command	1-20
Add Source and Include Folders	1-20
Manage Include File Sequence	1-20
Organize Layout of Polyspace User Interface	1-22
Create Your Own Layout	1-22
Save and Reset Layout	1-23
Customize Polyspace User Interface	1-24
Possible Customizations	1-24
Storage of Polyspace User Interface Customizations	1-26
Upload Results to Polyspace Access	1-28
Upload Results from Polyspace Desktop Client	1-28
Upload Results at Command Line	1-29
Results Upload Compatibility and Permissions	1-29

Run Polyspace Analysis with Windows or Linux Scripts

2

Run Polyspace Analysis from Command Line	2-2
Specify Sources and Analysis Options Directly	2-2
Specify Sources and Analysis Options in Text File	2-2
Create Options File from Build System	2-3
Options Files for Polyspace Analysis	2-5
What are Options Files	2-5
Specifying Options Files	2-5
Specifying Multiple Options Files	2-7
Modularize Polyspace Analysis by Using Build Command	2-8
Build Source Code	2-8
Create One Polyspace Options File for Full Build	2-10
Create Options File for Specific Binary in Build Command	2-11
Create One Options File Per Binary Created in Build Command	2-11
polyspace-configure Source Files Selection Syntax	2-14
Configure Polyspace Analysis Options in User Interface and Generate Scripts	2-16
Prerequisites	2-17
Generate Scripts from Configuration	2-17
Run Analysis with Generated Scripts	2-18

Run Polyspace Analysis with MATLAB Scripts

3

Integrate Polyspace with MATLAB and Simulink	3-2
Integrate Polyspace with MATLAB or Simulink from Same Release	3-2
Integrate Polyspace with MATLAB or Simulink Installation from Earlier Release	3-3
Check Integration Between MATLAB and Polyspace	3-4
Get Started with Polyspace Analysis by Using MATLAB	3-5
Prerequisites	3-5
Run Polyspace Analysis by Using MATLAB	3-5
Frequently Used MATLAB Functions	3-6
Run Polyspace Analysis by Using MATLAB Scripts	3-9
Prerequisites	3-9
Specify Multiple Source Files	3-9
Check for MISRA C:2012 Violations	3-10
Check for Specific Defects or Coding Rule Violations	3-10
Find Files That Do Not Compile	3-11
Run Analysis on Server	3-11

Compare Results from Different Polyspace Runs by Using MATLAB	
Scripts	3-13
Review Only New Results Compared to Last Run	3-13
Review New Results and Unreviewed Results from Last Run	3-14
Generate MATLAB Scripts from Polyspace User Interface	3-16
Prerequisites	3-16
Create Scripts from Polyspace Projects	3-16
Troubleshoot Polyspace Analysis from MATLAB	3-18
Prerequisites	3-18
Capture Polyspace Analysis Errors in Error Log	3-18

4 Offload Polyspace Analysis to Remote Servers from Desktop

Send Polyspace Analysis from Desktop to Remote Servers	4-2
Client-Server Workflow for Running Analysis	4-2
Prerequisites	4-3
Offload Analysis in Polyspace User Interface	4-3
Send Polyspace Analysis from Desktop to Remote Servers Using Scripts	
.....	4-6
Client-Server Workflow for Running Analysis	4-6
Prerequisites	4-7
Run Remote Analysis	4-7
Manage Remote Analysis	4-8
Sample Scripts for Remote Analysis	4-10

5 Run Polyspace Analysis in Simulink

Run Polyspace Analysis on Code Generated with Embedded Coder	5-2
Prerequisites	5-2
Generate and Analyze Code	5-2
Review Analysis Results	5-4
Annotate Blocks to Justify Issues	5-5
Changes in Polyspace Analysis Workflows in Simulink in R2019b	5-9
Code Verification Workflow in a Nutshell	5-9
Locate Pre-R2019b Menu Items in Simulink Toolstrip	5-10
Run Polyspace on Code Generated by Using Previous Releases of Simulink	5-12
Prerequisite	5-12
Run a Cross-Release Polyspace Analysis	5-12
Review Results	5-13

Run Polyspace Analysis on Code Generated from Simulink Model	5-15
Prerequisites	5-15
Open Model for Code Generation and Polyspace Analysis	5-15
Generate and Analyze Code	5-16
Review Analysis Results	5-17
Trace Errors Back to Model and Fix Them	5-17
Check for Coding Rule Violations	5-19
Annotate Blocks to Justify Results	5-19
Run Polyspace Analysis on Generated Code by Using Packaged Options	
Files	5-21
Generate and Package Polyspace Options Files	5-21
Run Polyspace Analysis by Using the Packaged Options Files	5-22
Run Polyspace Analysis on Custom Code in Simulink Models	5-24
Prerequisite	5-24
Analyze Custom Code	5-24
Review Analysis Results	5-25
Run Polyspace Analysis on S-Function Code	5-27
Prerequisites	5-27
S-Function Analysis Workflow	5-27
Compile S-Functions to Be Compatible with Polyspace	5-27
Example S-Function Analysis	5-27
Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow	
Charts	5-29
Prerequisites	5-29
C/C++ Function Called Once in Model	5-29
C/C++ Function Called Multiple Times in Model	5-33
Run Polyspace Analysis on Custom Code in C Function Block	5-37
Prerequisites	5-37
Open Model for Running Polyspace Analysis on Custom Code in C Function Block	5-37
Run Polyspace Analysis	5-38
Identify Issues in C Code	5-39
Fix Identified Issues	5-41
Recommended Model Configuration Parameters for Polyspace Analysis	5-43
Configure Advanced Polyspace Options in Simulink	5-45
Configure Options	5-45
Share and Reuse Configuration	5-47
How Polyspace Analysis of Generated Code Works	5-49
Default Polyspace Options for Code Generated with Embedded Coder .	5-50
Default Options	5-50
Constraint Specification	5-50
Recommended Polyspace options for Verifying Generated Code	5-51
Hardware Mapping Between Simulink and Polyspace	5-51

External Constraints on Polyspace Analysis of Generated Code	5-53
Extract External Constraints from Model	5-53
Storage Classes Supported for Constraint Extraction	5-54
Run Polyspace Analysis on Code Generated with TargetLink	5-56
Configure and Run Analysis	5-56
Review Analysis Results	5-57
Default Polyspace Options for Code Generated with TargetLink	5-58
TargetLink Support	5-58
Default Options	5-58
Lookup Tables	5-58
Data Range Specification	5-59
Code Generation Options	5-59
Troubleshoot Navigation from Code to Model	5-60
Links from Code to Model Do Not Appear	5-60
Links from Code to Model Do Not Work	5-60
Your Model Already Uses Highlighting	5-61
Polyspace Support of MATLAB and Simulink from Different Releases ..	5-62
Complete Integration	5-63
Cross-Release Integration	5-63
Navigate Back to Model	5-63

Run Polyspace Analysis in MATLAB Coder

6

Run Polyspace on C/C++ Code Generated from MATLAB Code	6-2
Prerequisites	6-2
Run Polyspace Analysis	6-2
Review Analysis Results	6-4
Run Analysis for Specific Design Range	6-5
Configure Advanced Polyspace Options in MATLAB Coder App	6-8
Configure Options	6-8
Share and Reuse Configuration	6-9

Run Polyspace Analysis in IDE Plugins

7

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

8

Specify Polyspace Analysis Options	8-2
Polyspace User Interface	8-2
Windows or Linux Scripts	8-2
MATLAB Scripts	8-3
Eclipse and Eclipse-based IDEs	8-3
Simulink	8-3
MATLAB Coder App	8-3

Configure Target and Compiler Options

9

Specify Target Environment and Compiler Behavior	9-2
Extract Options from Build Command	9-2
Specify Options Explicitly	9-3
C/C++ Language Standard Used in Polyspace Analysis	9-5
Supported Language Standards	9-5
Default Language Standard	9-5
C11 Language Elements Supported in Polyspace	9-7
C++11 Language Elements Supported in Polyspace	9-9
C++14 Language Elements Supported in Polyspace	9-12
C++17 Language Elements Supported in Polyspace	9-15
Provide Standard Library Headers for Polyspace Analysis	9-19
Requirements for Project Creation from Build Systems	9-20
Compiler Requirements	9-20
Build Command Requirements	9-21
Supported Keil or IAR Language Extensions	9-23
Special Function Register Data Type	9-23
Keywords Removed During Preprocessing	9-24
Remove or Replace Keywords Before Compilation	9-25
Remove Unrecognized Keywords	9-25
Remove Unrecognized Function Attributes	9-27
Gather Compilation Options Efficiently	9-28

Specify External Constraints	10-2
Create Constraint Template	10-2
Create Constraint Template from Code Prover Analysis Results	10-4
Update Existing Template	10-4
Specify Constraints in Code	10-5
 External Constraints for Polyspace Analysis	 10-7
Constraint Specification Limitations	10-11
 Constrain Global Variable Range	 10-13
User Interface (Desktop Products Only)	10-13
Command Line	10-14
 Constrain Function Inputs	 10-16
User Interface (Desktop Products Only)	10-16
Command Line	10-17
 XML File Format for Constraints	 10-19
Syntax Description — XML Elements	10-19
Valid Modes and Default Values	10-23

Analyze Multitasking Programs in Polyspace	11-2
Configure Analysis	11-2
Review Analysis Results	11-3
 Auto-Detection of Thread Creation and Critical Section in Polyspace	 11-5
Multitasking Routines that Polyspace Can Detect	11-5
Example of Automatic Thread Detection	11-7
Naming Convention for Automatically Detected Threads	11-10
Limitations of Automatic Thread Detection	11-11
 Configuring Polyspace Multitasking Analysis Manually	 11-16
Specify Options for Multitasking Analysis	11-16
Adapt Code for Code Prover Multitasking Analysis	11-16
 Protections for Shared Variables in Multitasking Code	 11-20
Detect Unprotected Access	11-20
Protect Using Critical Sections	11-21
Protect Using Temporally Exclusive Tasks	11-22
Protect Using Priorities	11-22
Protect By Disabling Interrupts	11-23
 Define Atomic Operations in Multitasking Code	 11-24
Nonatomic Operations	11-24

What Polyspace Considers as Nonatomic	11-24
Define Specific Operations as Atomic	11-25
Define Preemptable Interrupts and Nonpreemptable Tasks	11-27
Emulating Task Priorities	11-27
Examples of Task Priorities	11-27
Further Explorations	11-28
Define Critical Sections with Functions That Take Arguments	11-30
Polyspace Assumption on Functions Defining Critical Sections	11-30
Adapt Polyspace Analysis to Lock and Unlock Functions with Arguments	11-30

Configure Coding Rules Checking and Code Metrics Computation

12

Check for Coding Standard Violations	12-2
Configure Coding Rules Checking	12-2
Review Coding Rule Violations	12-6
Generate Reports	12-8
Avoid Violations of MISRA C:2012 Rules 8.x	12-9
Reduce Software Complexity by Using Polyspace Checkers	12-12
Configure Thresholds for Software Complexity Checkers	12-12
Identify and Reduce Software Complexity	12-13
Software Quality Objective Subsets (C:2004)	12-16
Rules in SQO-Subset1	12-16
Rules in SQO-Subset2	12-17
Software Quality Objective Subsets (AC AGC)	12-20
Rules in SQO-Subset1	12-20
Rules in SQO-Subset2	12-20
Software Quality Objective Subsets (C:2012)	12-23
Guidelines in SQO-Subset1	12-23
Guidelines in SQO-Subset2	12-24
Software Quality Objective Subsets (C++)	12-26
SQO Subset 1 - Direct Impact on Selectivity	12-26
SQO Subset 2 - Indirect Impact on Selectivity	12-27
Coding Rule Subsets Checked Early in Analysis	12-32
MISRA C:2004 and MISRA AC AGC Rules	12-32
MISRA C:2012 Rules	12-39
Create Custom Coding Rules	12-47
User Interface (Desktop Products Only)	12-47
Command Line	12-48

Compute Code Complexity Metrics	12-49
Impose Limits on Metrics (Desktop Products Only)	12-49
Impose Limits on Metrics (Server and Access products)	12-51
HIS Code Complexity Metrics	12-52
Project	12-52
File	12-52
Function	12-52

Coding Rule Sets and Concepts

13

Polyspace MISRA C:2004 and MISRA AC AGC Checkers	13-2
MISRA C:2004 and MISRA AC AGC Coding Rules	13-3
Supported MISRA C:2004 and MISRA AC AGC Rules	13-3
Troubleshooting	13-3
List of Supported Coding Rules	13-3
Unsupported MISRA C:2004 and MISRA AC AGC Rules	13-36
Polyspace MISRA C:2012 Checkers	13-38
Essential Types in MISRA C:2012 Rules 10.x	13-39
Categories of Essential Types	13-39
How MISRA C:2012 Uses Essential Types	13-39
Unsupported MISRA C:2012 Guidelines	13-41
Polyspace MISRA C++ Checkers	13-42
Unsupported MISRA C++ Coding Rules	13-43
Language Independent Issues	13-43
General	13-44
Lexical Conventions	13-44
Expressions	13-44
Declarations	13-44
Classes	13-45
Templates	13-45
Exception Handling	13-45
Library Introduction	13-45
Polyspace JSF AV C++ Checkers	13-47
JSF AV C++ Coding Rules	13-48
Supported JSF C++ Coding Rules	13-48
Unsupported JSF++ Rules	13-66

Choose Specific Bug Finder Defect Checkers	14-2
User Interface (Desktop Products Only)	14-2
Command Line	14-2
Modify Default Behavior of Bug Finder Checkers	14-4
Defect Checkers	14-4
Coding Standard Checkers	14-6
Flag Deprecated or Unsafe Functions Using Bug Finder Checkers	14-9
Identify Need for Extending Checker	14-9
Extend Checker	14-10
Checkers That Can Be Extended	14-10
Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries	14-11
Identify Need for Extending Checker	14-11
Extend Checker	14-11
Checkers That Can Be Extended	14-12
Extend Bug Finder Checkers to Find Defects from Specific System Input Values	14-13
Identify Need for Extending Checker	14-13
Extend Checker	14-13
Checkers That Can Be Extended	14-14
Extend Concurrency Defect Checkers to Unsupported Multithreading Environments	14-16
Identify Need for Extending Checker	14-16
Extend Checker	14-17
Checkers That Can Be Extended	14-17
Extend Checkers for Initialization to Check Function Arguments Passed by Pointers	14-19
Identify Need for Existing Checker	14-19
Extend Checker	14-19
Checkers That Can Be Extended	14-20
Prepare Checkers Configuration for Polyspace Bug Finder Analysis ..	14-21
Identify Checkers to Enable	14-21
Create Checkers Configuration Files	14-21
Short Names of Bug Finder Defect Checkers	14-26
Bug Finder Defect Groups	14-40
C++ Exceptions	14-40
Concurrency	14-40
Cryptography	14-41
Data flow	14-41
Dynamic Memory	14-42
Good Practice	14-42
Numerical	14-42

Object Oriented	14-42
Performance	14-43
Programming	14-43
Resource Management	14-43
Static Memory	14-43
Security	14-44
Tainted data	14-44
Sources of Tainting in a Polyspace Analysis	14-45
Sources of Tainted Data	14-45
Impact of Tainted Data Defects	14-45
Polyspace Bug Finder Defects Checkers Enabled by Default	14-48
Bug Finder Results Found in Fast Analysis Mode	14-53
Polyspace Bug Finder Defects	14-53
MISRA C:2004 and MISRA AC AGC Rules	14-56
MISRA C:2012 Rules	14-62
MISRA C++ 2008 Rules	14-68
CWE Coding Standard and Polyspace Results	14-78
CWE and Polyspace Bug Finder	14-78
Find CWE IDs from Polyspace Results	14-78
Mapping Between CWE Identifiers and Polyspace Results	14-78
Mapping Between CWE-658 or 659 and Polyspace Results	14-103
CWE-658: Weaknesses in Software Written in C	14-103
CWE-659: Weaknesses in Software Written in C++	14-109

Configure Comment Import from Previous Results

15

Import Review Information from Previous Polyspace Analysis	15-2
Automatic Import from Last Analysis	15-2
Import from Another Analysis Result	15-2
Import Algorithm	15-3
View Imported Review Information That Does Not Apply	15-4
Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results	15-6
Mapping Multiple MISRA C: 2004 Annotations to the Same MISRA C: 2012 Result	15-7

Interpret Polyspace Bug Finder Results

16

Interpret Bug Finder Results in Polyspace Desktop User Interface	16-2
Interpret Result Details Message	16-3
Find Root Cause of Result	16-4

Investigate the Cause of Empty Results List	16-7
Dashboard	16-9
Concurrency Modeling	16-13
Results List	16-15
Source	16-18
Tooltips	16-18
Examine Source Code	16-18
Expand Macros	16-19
Manage Multiple Files in Source Pane	16-21
View Code Block	16-22
Call Hierarchy	16-23
Result Details	16-25

Fix or Comment Polyspace Results

17

Address Polyspace Results Through Bug Fixes or Justifications	17-2
Add Review Information to Results File	17-2
Comment or Annotate in Code	17-3
Annotate Code and Hide Known or Acceptable Results	17-6
Code Annotation Syntax	17-6
Syntax Examples	17-10
Short Names of Code Complexity Metrics	17-13
Project Metrics	17-13
File Metrics	17-13
Function Metrics	17-13
Annotate Code for Known or Acceptable Results (Not Recommended)	17-15
Add Annotations from the Polyspace Interface	17-15
Add Annotations Manually	17-16
Define Custom Annotation Format	17-19
Define Annotation Syntax Format	17-21
Map Your Annotation to the Polyspace Annotation Syntax	17-24
Define Multiple Custom Annotation Syntaxes	17-25
Annotation Description Full XML Template	17-27
Example	17-30

18

Filter and Group Results in Polyspace Desktop User Interface	18-2
Filter Results	18-3
Group Results	18-7
Classification of Defects by Impact	18-9
High Impact Defects	18-9
Medium Impact Defects	18-11
Low Impact Defects	18-16

Generate Reports from Polyspace Results

19

Generate Reports	19-2
Generate Reports from User Interface	19-2
Generate Reports from Command Line	19-3
Export Polyspace Analysis Results	19-5
Export Results to Text File	19-5
Export Results to MATLAB Table	19-5
Export Results to JSON Format	19-5
View Exported Results	19-6
Export Polyspace Analysis Results to Excel by Using MATLAB Scripts	19-7
Report Result Summary and Details in One Worksheet	19-7
Control Formatting of Excel Report	19-8
Visualize Bug Finder Analysis Results in MATLAB	19-9
Export Results to MATLAB Table	19-9
Generate Graphs from Results and Include in Report	19-9
Customize Existing Bug Finder Report Template	19-13
Prerequisites	19-13
View Components of Template	19-13
Change Components of Template	19-14

Software Quality with Polyspace Metrics

20

Upload Results to Polyspace Metrics	20-2
Manually Upload Results	20-2
Automatically Upload Results (Batch Analysis Only)	20-3
View Projects in Polyspace Metrics	20-4
Upload Results	20-4

Open Metrics Interface	20-4
Review Metrics	20-5
Compare Metrics Between Results	20-6
Polyspace Metrics Interface	20-7
Compare Metrics Against Software Quality Objectives	20-11
Apply Predefined Objectives to Metrics	20-11
Bug Finder Quality Objective Levels	20-12
Customize Software Quality Objectives	20-15
Web Browser Requirements for Polyspace Metrics	20-19
Additional Considerations	20-19
View Results List in Polyspace Metrics	20-20
Open Polyspace Metrics	20-20
View Results List	20-21
Download Results	20-22

Troubleshooting in Polyspace Bug Finder

21

License Error -4,0	21-2
Issue	21-2
Possible Cause: Another Polyspace Instance Running	21-2
Possible Cause: Prior Polyspace Run in Simulink or MATLAB Coder	21-2
View Error Information When Analysis Stops	21-3
View Error Information in User Interface	21-3
View Error Information in Log File	21-4
Contact Technical Support About Issues with Running Polyspace	21-6
Provide System Information	21-6
Provide Information About the Issue	21-6
Compiler Not Supported for Project Creation from Build Systems	21-9
Issue	21-9
Cause	21-9
Solution	21-9
Slow Build Process When Polyspace Traces the Build	21-15
Issue	21-15
Cause	21-15
Solution	21-15
Check if Polyspace Supports Build Scripts	21-16
Issue	21-16
Possible Cause	21-16
Solution	21-16
Troubleshooting Project Creation from MinGW Build	21-18
Issue	21-18
Cause	21-18

Solution	21-18
Troubleshooting Project Creation from Visual Studio Build	21-19
Polyspace Cannot Find the Server	21-20
Message	21-20
Possible Cause	21-20
Solution	21-20
Job Manager Cannot Write to Database	21-21
Message	21-21
Possible Cause	21-21
Workaround	21-21
Undefined Identifier Error	21-22
Issue	21-22
Possible Cause: Missing Files	21-22
Possible Cause: Unrecognized Keyword	21-22
Possible Cause: Declaration Embedded in #ifdef Statements	21-23
Possible Cause: Project Created from Non-Debug Build	21-23
Unknown Function Prototype Error	21-25
Issue	21-25
Cause	21-25
Solution	21-25
Error Related to #error Directive	21-26
Issue	21-26
Cause	21-26
Solution	21-26
Large Object Error	21-27
Issue	21-27
Cause	21-27
Solution	21-27
Errors Related to Generic Compiler	21-29
Issue	21-29
Cause	21-29
Solution	21-29
Errors Related to Keil or IAR Compiler	21-30
Missing Identifiers	21-30
Errors Related to Diab Compiler	21-31
Issue	21-31
Cause	21-31
Solution	21-31
Errors Related to Green Hills Compiler	21-33
Issue	21-33
Cause	21-33
Solution	21-33

Errors Related to TASKING Compiler	21-35
Issue	21-35
Cause	21-35
Solution	21-35
Errors from Conflicts with Polyspace Header Files	21-37
Issue	21-37
Cause	21-37
Solution	21-37
Errors from Using Namespace std Without Prefix	21-38
Issue	21-38
Cause	21-38
Solution	21-38
Errors from Assertion or Memory Allocation Functions	21-39
Issue	21-39
Cause	21-39
Solution	21-39
Errors from In-Class Initialization (C++)	21-40
Errors from Double Declarations of Standard Template Library Functions (C++)	21-41
Errors Related to GNU Compiler	21-42
Issue	21-42
Cause	21-42
Solution	21-42
Errors Related to Visual Compilers	21-43
Import Folder	21-43
pragma Pack	21-43
C++/CLI	21-44
Eclipse Java Version Incompatible with Polyspace Plug-in	21-45
Issue	21-45
Cause	21-45
Solution	21-45
Coding Standard Violations Not Displayed	21-46
Issue	21-46
Possible Cause: Rule Checker Not Enabled	21-46
Possible Cause: Rule Violations in Header Files	21-46
Possible Cause: Rule Violations in Macros	21-46
Possible Cause: Compilation Errors	21-47
Possible Cause: Code Prover Analysis with Lower Verification Level ...	21-47
Insufficient Memory During Report Generation	21-48
Message	21-48
Possible Cause	21-48
Solution	21-48

Error or Slow Runs from Disk Defragmentation and Anti-virus Software	21-49
.....	21-49
Issue	21-49
Possible Cause	21-49
Solution	21-49
SQLite I/O Error	21-51
Issue	21-51
Cause	21-51
Solution	21-51
Errors with Temporary Files	21-52
No Access Rights	21-52
No Space Left on Device	21-52
Cannot Open Temporary File	21-52
Resolve -xml-annotations-description Errors	21-54
Issue	21-54
Possible Solutions	21-54

Run Polyspace Analysis on Desktop

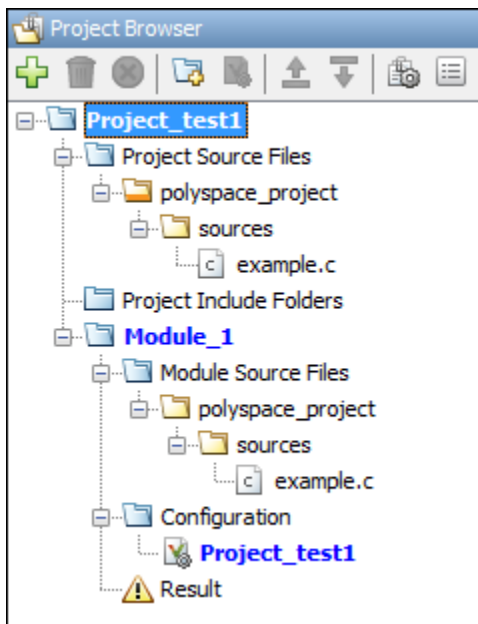
- “Add Source Files for Analysis in Polyspace User Interface” on page 1-2
- “Run Polyspace Analysis on Desktop” on page 1-7
- “Project and Results Folder Contents” on page 1-11
- “Storage of Temporary Files” on page 1-13
- “Create Project Using Visual Studio Information” on page 1-14
- “Create Project Using Configuration Template” on page 1-16
- “Update Polyspace Project” on page 1-19
- “Organize Layout of Polyspace User Interface” on page 1-22
- “Customize Polyspace User Interface” on page 1-24
- “Upload Results to Polyspace Access” on page 1-28

Add Source Files for Analysis in Polyspace User Interface

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). 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.

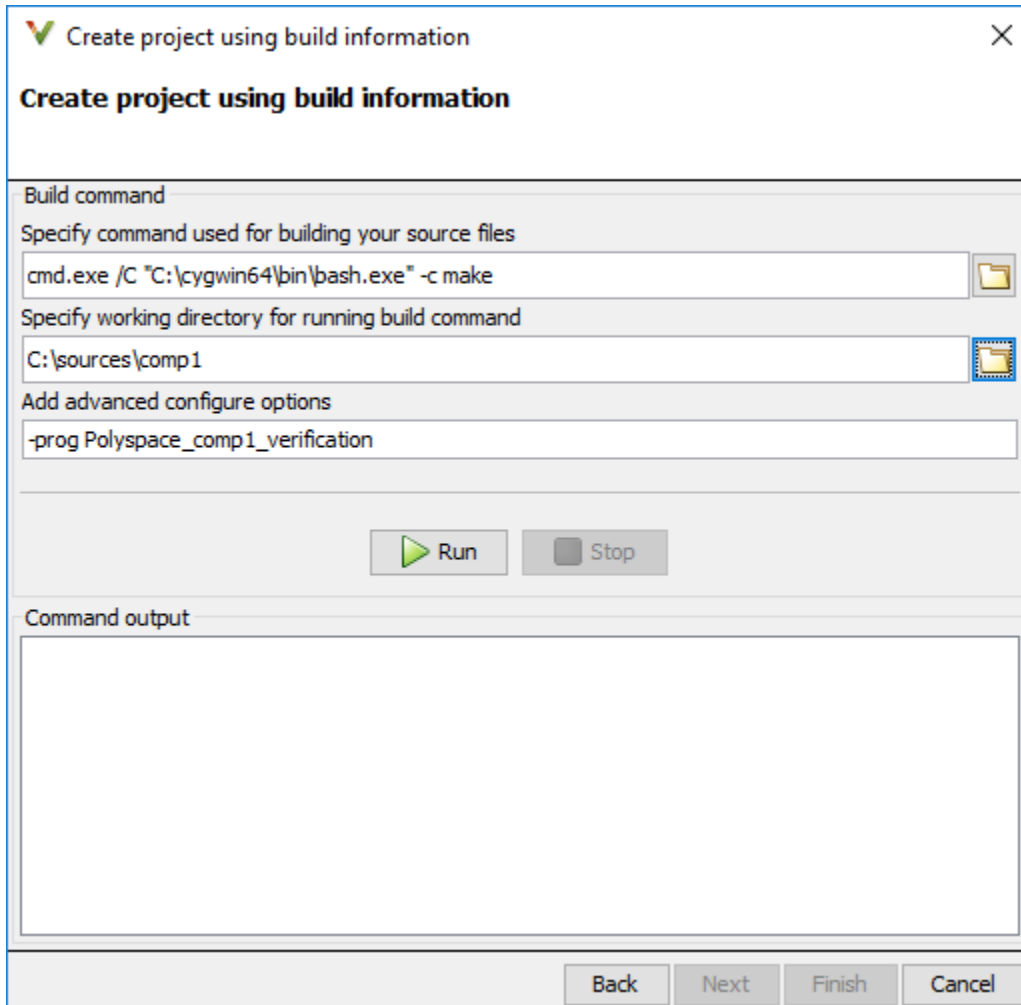
Note that when you reopen a project, the source file paths are computed based on the project location. For instance, suppose you commit the Polyspace project to a version control system along with your source files. When you check out the project from your version control system and open a local copy 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 Project Using Visual Studio Information" on page 1-14.

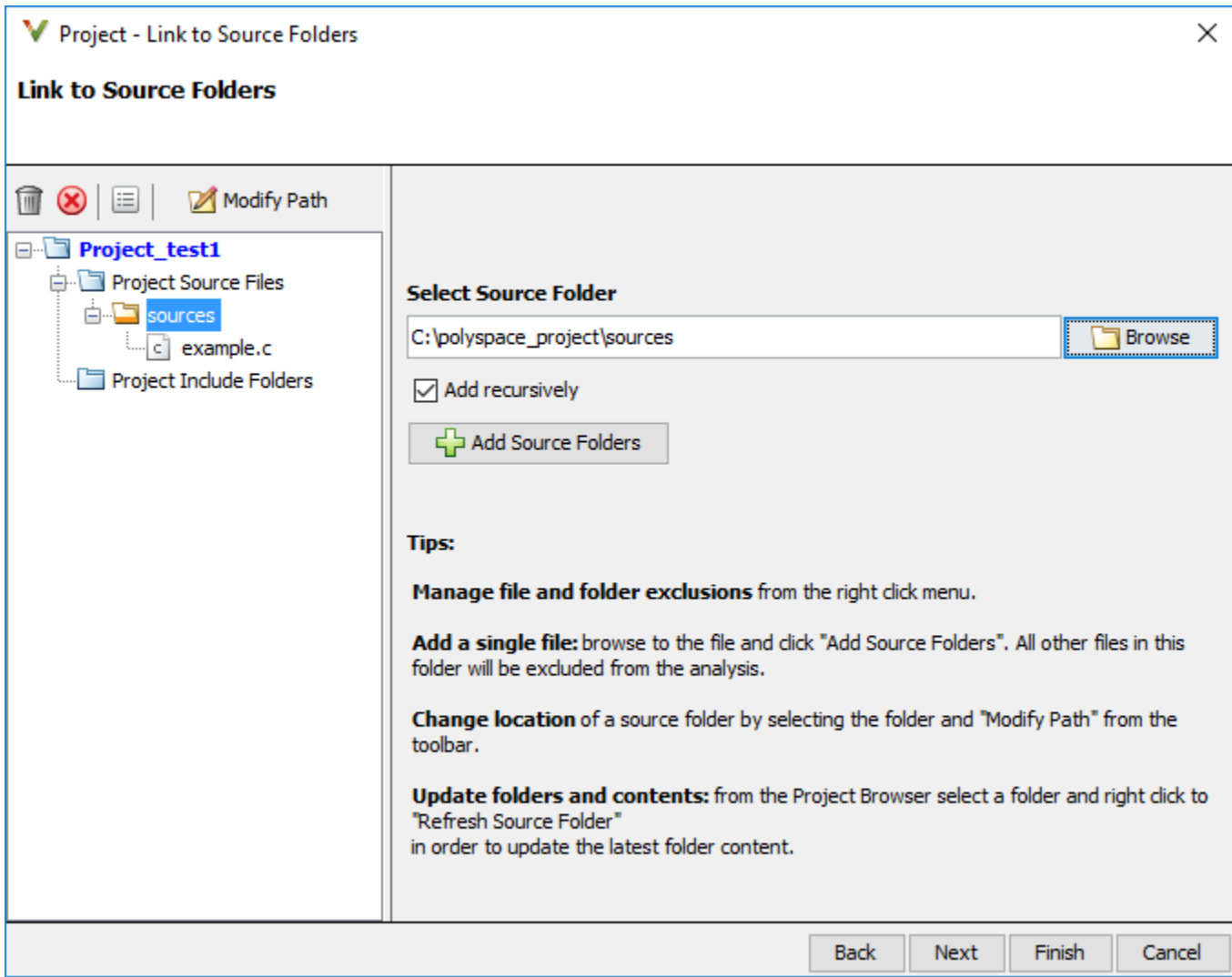
When you create a project from your build command, the **Project Browser** pane displays your source folders but not the include folders. In case you want to verify that your include folders were extracted, open the project file (with extension `.psprj`) in a text editor.

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 9-19.

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.

See Also

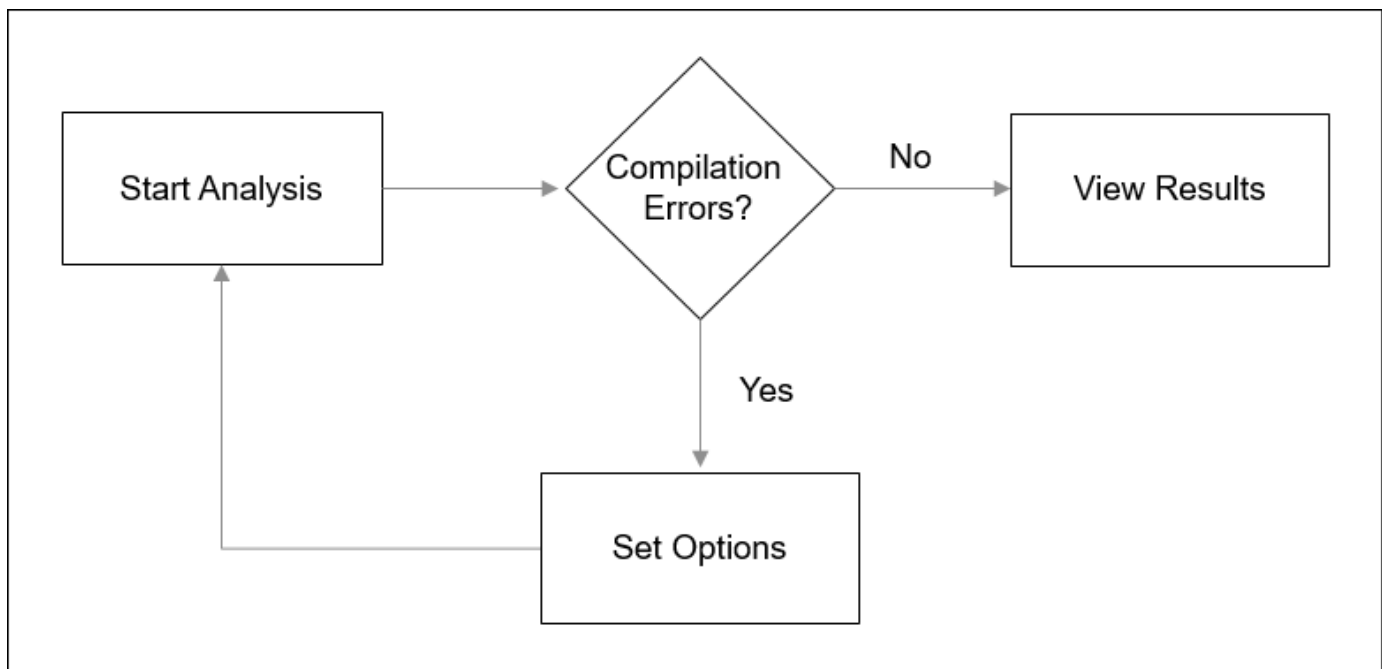
More About

- “Run Polyspace Analysis on Desktop” on page 1-7
- “Create Project Using Visual Studio Information” on page 1-14
- “Provide Standard Library Headers for Polyspace Analysis” on page 9-19

Run Polyspace Analysis on Desktop

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 1-2, 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**.

1 Run Polyspace Analysis on Desktop

➤ Select product.
➤ Start/stop analysis.

Set options as needed:

- Target & Compiler
- Macros
- Environment Settings

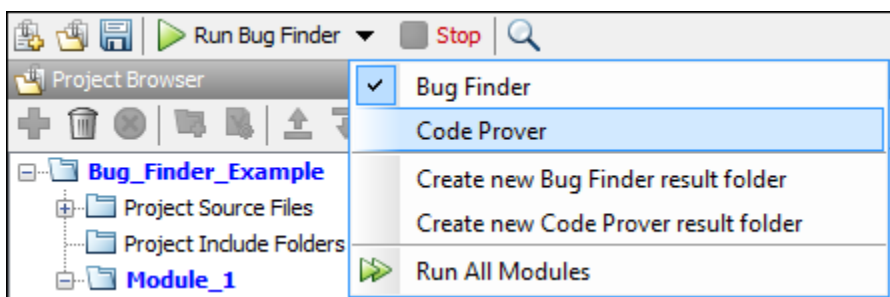
➤ Monitor progress
➤ Check for warnings and errors.

Select error message for more details.

The screenshot shows the Polyspace IDE interface. The 'Project Browser' on the left displays a project structure with 'Bug_Finder_Example' selected. The 'Configuration' window on the right shows the 'Target & Compiler' settings, including 'Source code language' set to 'C', 'Compiler' set to 'gnu4.6', and 'Target processor type' set to 'x86_64'. Below the configuration, the 'Output Summary' window shows the progress of the verification running, with a progress bar indicating 'Compiler: 93%' and 'Total: 15%'. A table of messages is displayed, including warnings and errors. One error message is highlighted, and a callout box indicates that clicking on it will show more details.

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, select **Create new Bug Finder result folder**.



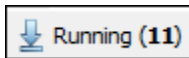
The results are stored in subfolders `Module_1`, `Module_2`, and so on in the project folder. To find the physical location of the project folder, right-click a project on the **Project Browser** pane and select **Open Folder with File Manager**.

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.

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

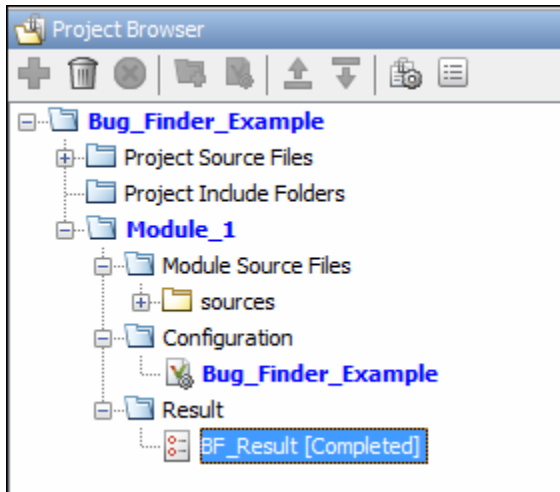
If compilation errors occur, the analysis continues on the remaining files that do compile. The **Dashboard** pane shows that some files did not compile and links to the **Output Summary** pane for details. The **Output Summary** pane shows compilation errors with a  icon.

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 “Project and Results Folder Contents” on page 1-11.

See Also

More About

- “Run Polyspace Analysis from Command Line” on page 2-2
- “Run Polyspace Analysis by Using MATLAB Scripts” on page 3-9
- “Interpret Bug Finder Results in Polyspace Desktop User Interface” on page 16-2
- “Address Polyspace Results Through Bug Fixes or Justifications” on page 17-2
- “Filter and Group Results in Polyspace Desktop User Interface” on page 18-2

Project and Results Folder Contents

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

Storage of Temporary Files

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:

- “Errors with Temporary Files” on page 21-52
- “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`

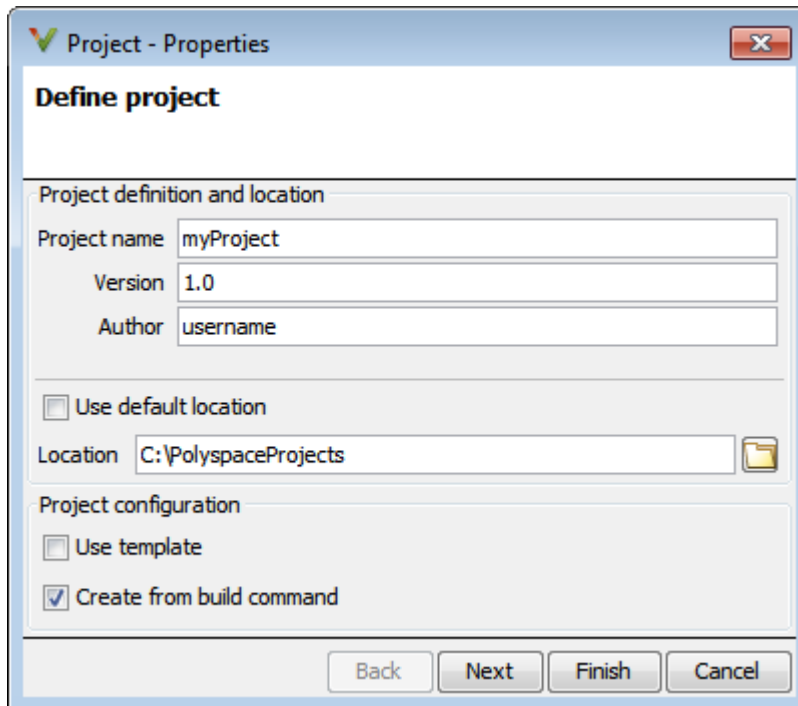
If one of these variables is defined, Polyspace uses that path for storing temporary files. If these environment variables are not defined, Polyspace stores temporary files in:


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

Create Project Using Visual Studio Information

To create a Polyspace project, you can trace your Visual Studio build.

- 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**.

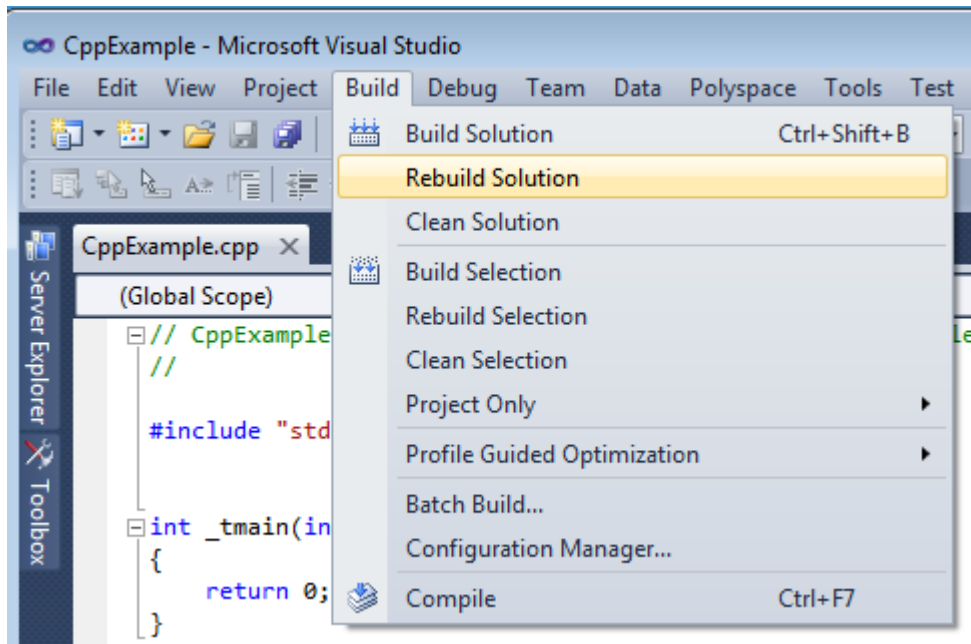


- 3 In the field **Specify command used for building your source files**, enter the full path to the Visual Studio executable. For instance, "C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\devenv.exe".
- 4 In the field **Specify working directory for running build command**, enter a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

More About

- “Troubleshooting Project Creation from Visual Studio Build” on page 21-19

Create Project Using Configuration Template

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

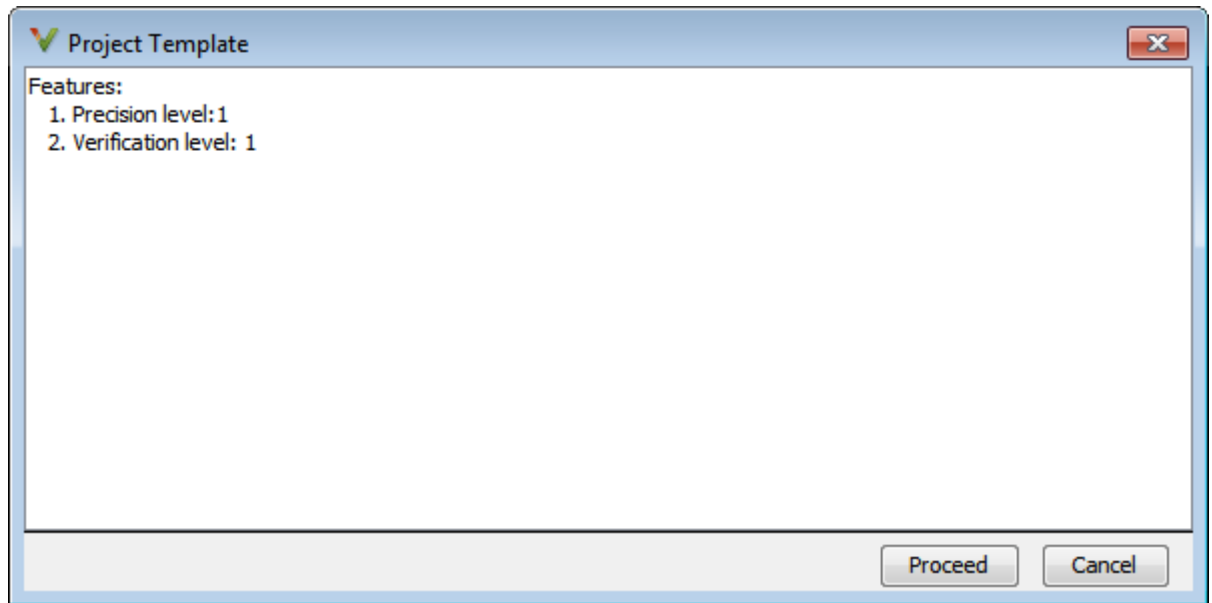
This example shows how to save a configuration from an existing project and create a new project using the saved configuration.


- 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.

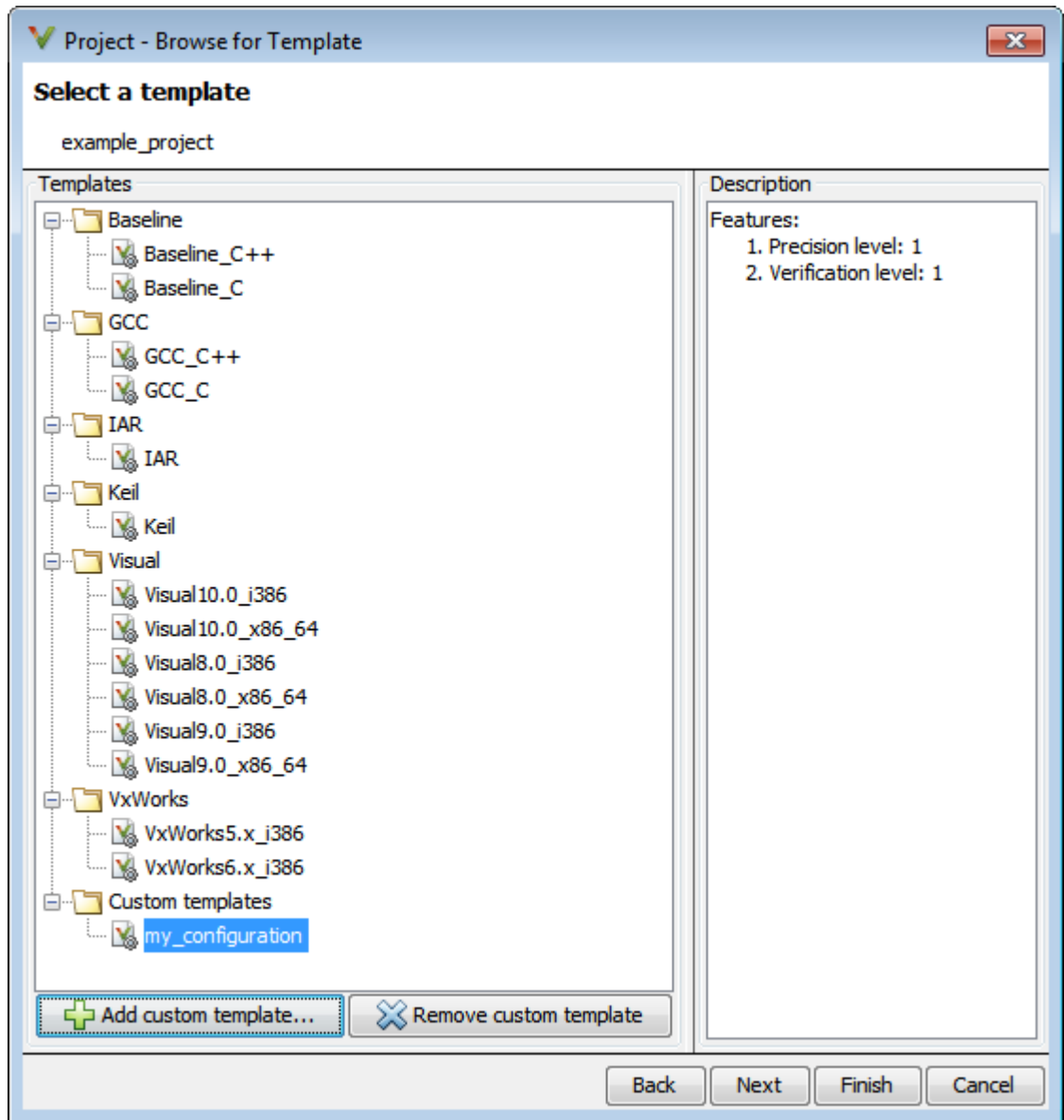
Suppose you create a Code Prover configuration template that runs Code Prover analysis to a precision level of 1 and a verification level of 1. See:

- Precision level (-0) (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 .
 - 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.



See Also

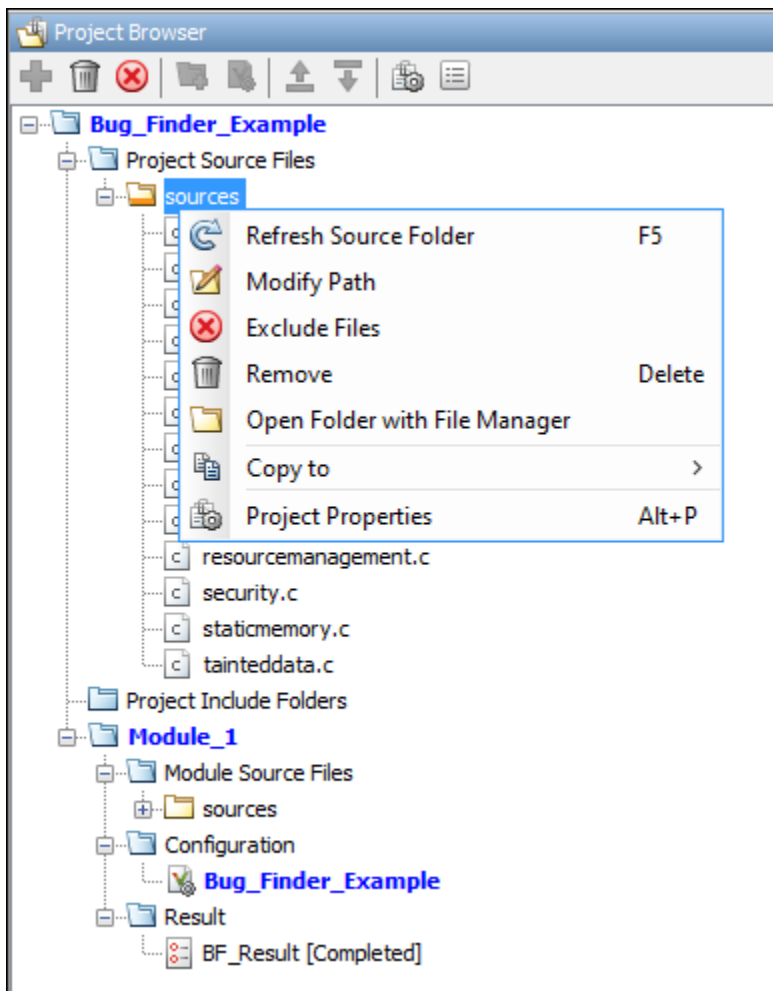
More About

- "Specify Polyspace Analysis Options" on page 8-2
- "Analysis Options in Polyspace Bug Finder"

Update Polyspace Project


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**.

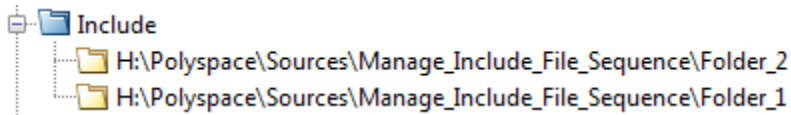
Before running an analysis, you must copy the source files to a module. Select the source files that you want to copy. To select multiple files together, press the **Ctrl** key while selecting the files. Right-click your selection. Select **Copy to > Module_n**. *n* is the module number.



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 User Interface” on page 1-2

Organize Layout of Polyspace 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.

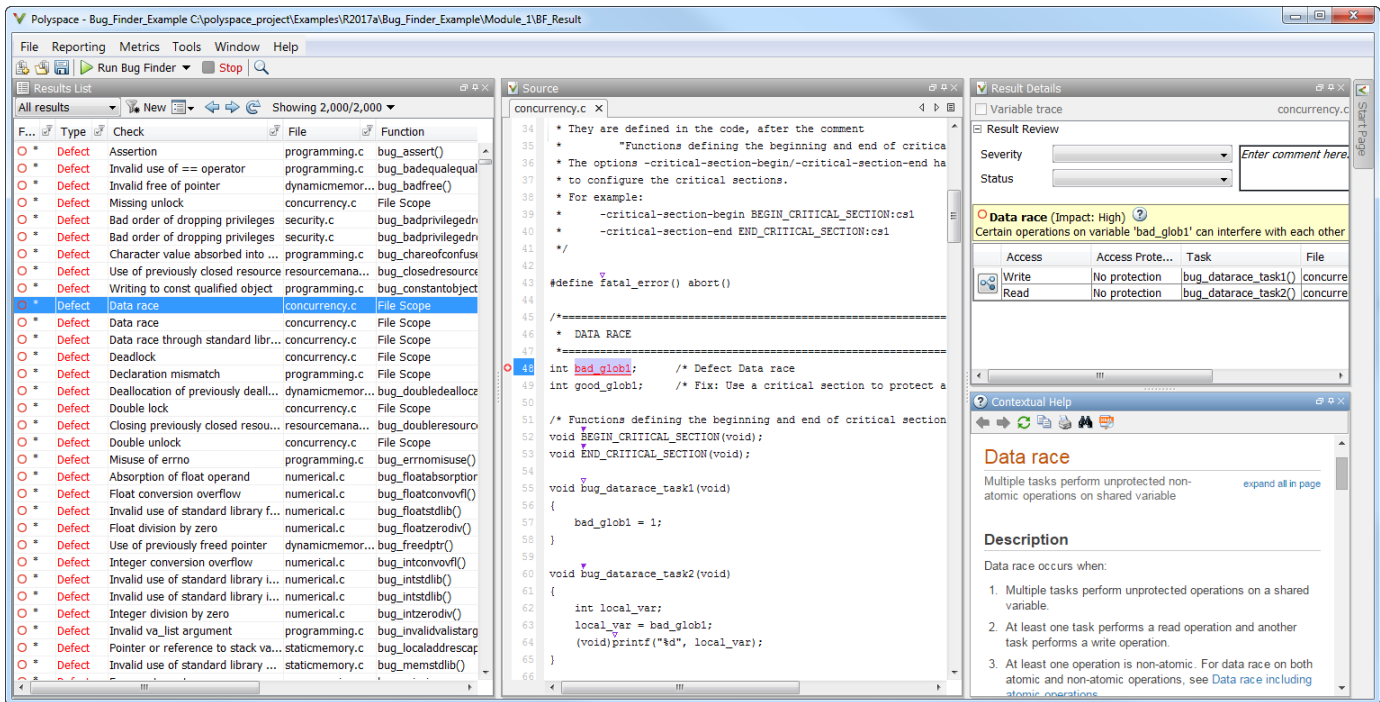
For instance, if **Project Browser** and **Results List** are tabbed on the same pane, dragging out **Project Browser** creates a pane with only **Project Browser** on it and floats this new pane.

- 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 User Interface” on page 1-24
- “Organize Layout of Polyspace User Interface” on page 1-22

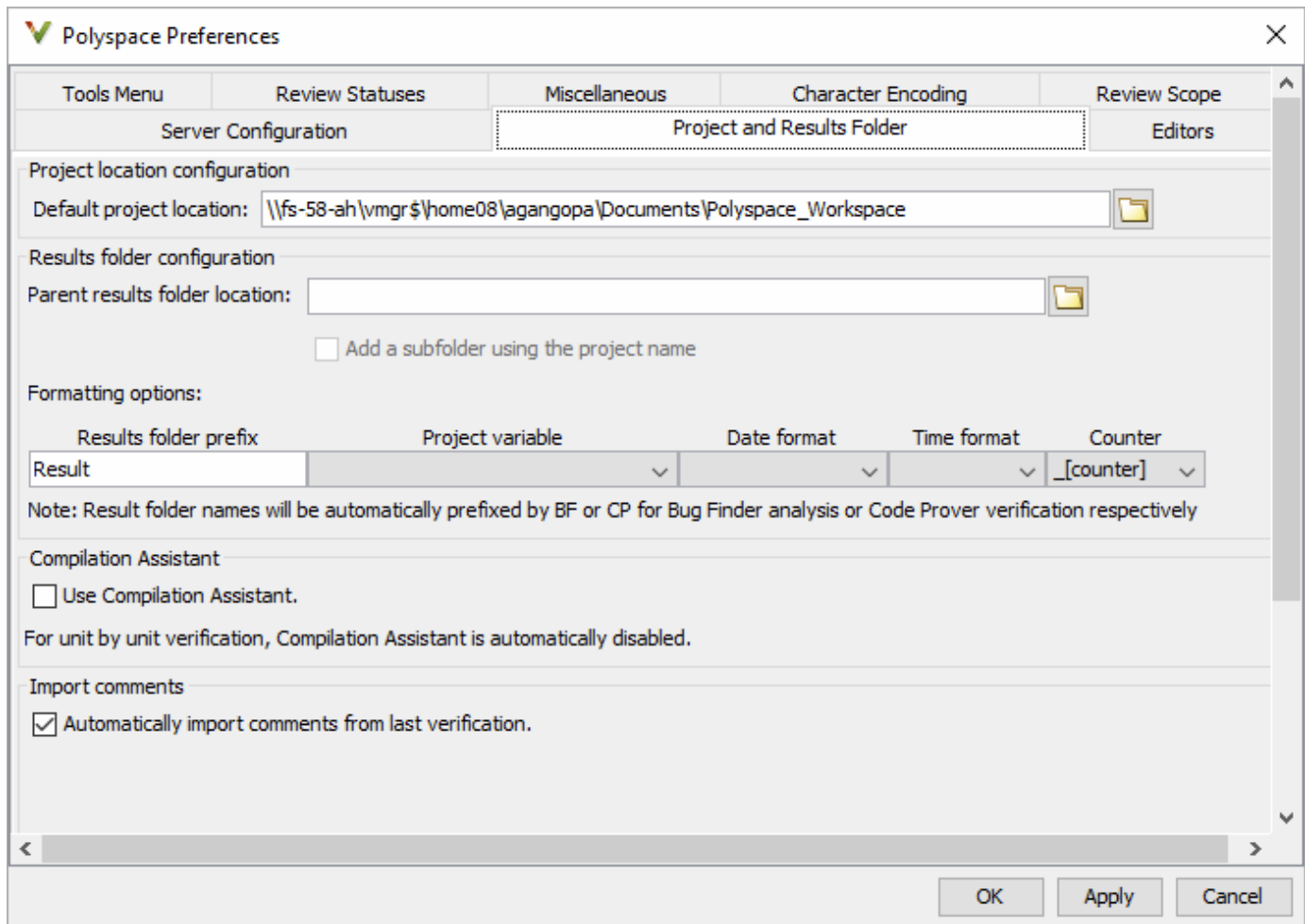
Customize Polyspace User Interface

In this section...

“Possible Customizations” on page 1-24

“Storage of Polyspace User Interface Customizations” on page 1-26

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.

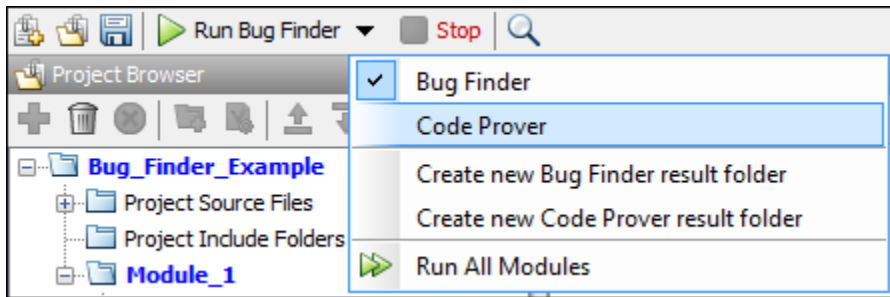
For console-based text editors, you must create a terminal. For example, to specify `vi`:

- 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

When you run an analysis, you can overwrite the results of the previous run or create a new results folder.



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, select the **Project and Results Folder** tab. 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

Create Custom Review Status

When reviewing Polyspace results, you can assign a status such as To fix or Justified. See “Address Polyspace Results Through Bug Fixes or Justifications” on page 17-2.

You can create your own statuses to assign. To create a new status, select the **Review Statuses** tab.

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.

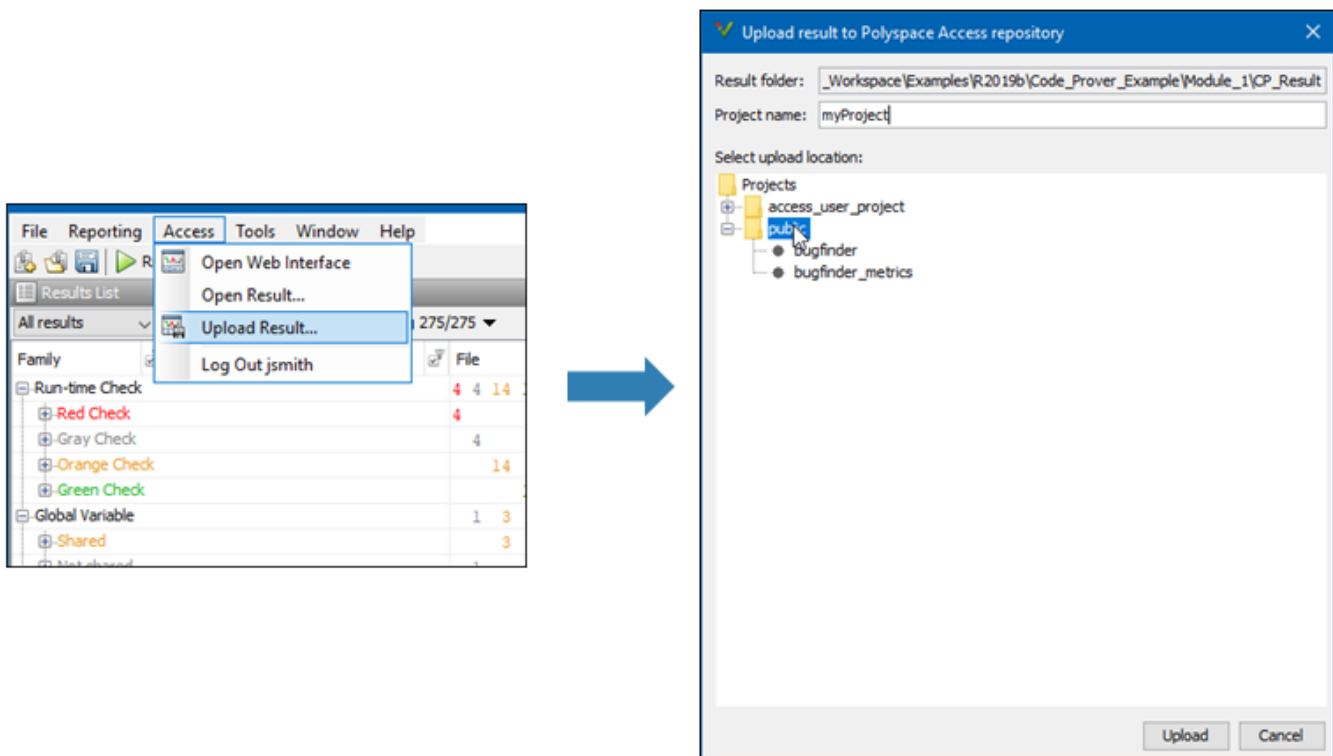
Polyspace assigns a unique run ID to each analysis run that you upload and increments the run ID with each upload to any project. If you use an automation tool such as Jenkins to upload results, the Polyspace Access run ID is not related to the tool job ID.

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” (Polyspace Bug Finder Access).

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 “Open Polyspace Access Results in a Desktop Interface” (Polyspace Bug Finder Access).

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 -host hostName -port port -upload BF_results
```

The command prompts you for your Polyspace Access login credentials, then uploads the results to the **public** folder of the Polyspace Access database. To upload results to a different folder, use the `-parent-folder` option. `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” (Polyspace Bug Finder Access).

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” (Polyspace Bug Finder Access).

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” (Polyspace Bug Finder Access).

See Also

polyspace-access

More About

- “Register Polyspace Desktop User Interface” (Polyspace Bug Finder Access)
- “Interpret Results” (Polyspace Bug Finder Access)
- “Manage Results” (Polyspace Bug Finder Access)

Run Polyspace Analysis with Windows or Linux Scripts

- “Run Polyspace Analysis from Command Line” on page 2-2
- “Options Files for Polyspace Analysis” on page 2-5
- “Modularize Polyspace Analysis by Using Build Command” on page 2-8
- “polyspace-configure Source Files Selection Syntax” on page 2-14
- “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 2-16

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\R2021a`. 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:

- “Analysis Options in Polyspace Bug Finder”
- “Analysis Options in Polyspace Code Prover” (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-bug-finder` | `polyspace-code-prover` | `polyspace-configure`

More About

- “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 2-16

- “Modularize Polyspace Analysis by Using Build Command” on page 2-8

External Websites

- Set up Continuous Code Verification with Jenkins

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:

- *Build options:*

You can extract build options from existing artifacts such as build commands and JSON compilation database, or collect all build options in an options file. You can specify this options file in the appropriate extension setting:

- 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**)
- *Checkers:*

You can specify checkers using a checkers selection wizard. For details, see “Setting Checkers in Polyspace as You Code” (Polyspace Bug Finder Access).

- *Other remaining options:*

All remaining options can be collected in a second options file that goes into the appropriate extension setting:

- 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 2-7.

For more information on IDE extensions, see:

- “Configure Polyspace as You Code Extension in Visual Studio” (Polyspace Bug Finder Access)
- “Configure Polyspace as You Code Extension in Visual Studio Code” (Polyspace Bug Finder Access)
- “Configure Polyspace as You Code Plugin in Eclipse” (Polyspace Bug Finder Access)

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, you can enter the following in the **Other** field:

```
-options-file 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 2-2
- “Analysis Options in Polyspace Bug Finder”

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\R2021a` or `C:\Program Files\Polyspace Server\R2021a`.

Build Source Code

Inspect the makefile. The makefile creates four binaries:

```
CC := gcc
LD := ld

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)

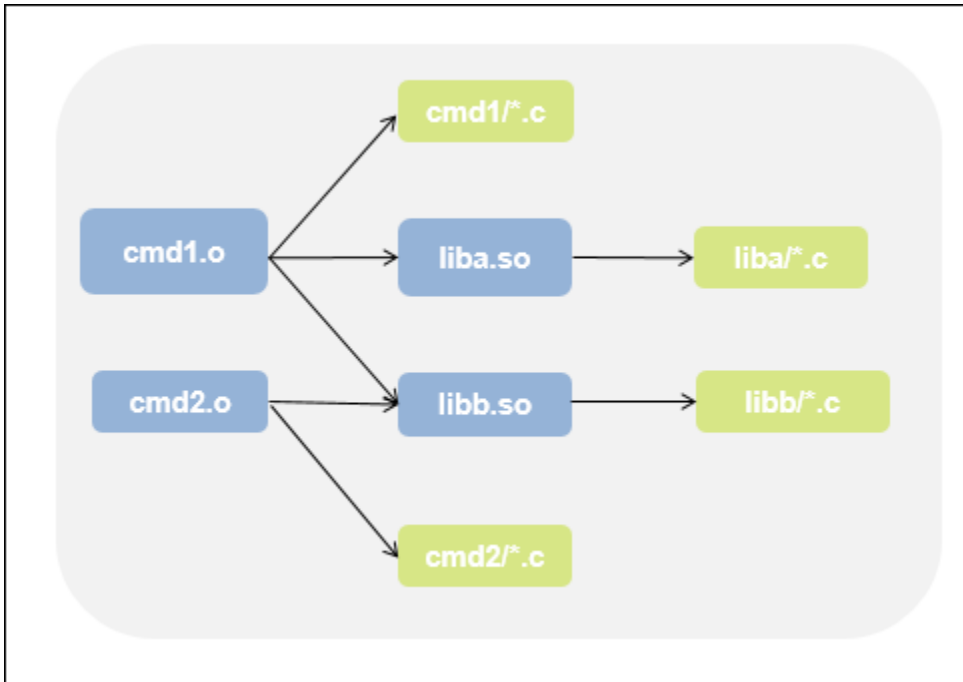
cmd2: libb
    $(CC) -c $(CMD2_SOURCES)
    $(LD) -o $@ $(CMD2_OBJ) $(LIBB_SOBJ)

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

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

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

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 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 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 2-11.

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-bug-finder` | `polyspace-configure`

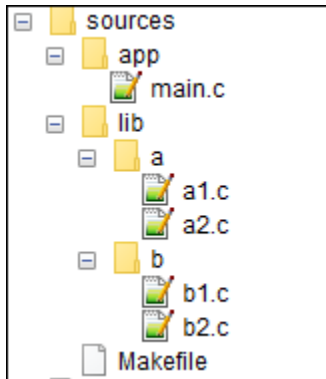
More About

- “Run Polyspace Analysis from Command Line” on page 2-2

polyspace-configure Source Files Selection Syntax

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.

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.	<code>-include-sources "b?.c"</code> matches: <code>/sources/lib/b/b1.c</code> <code>/sources/lib/b/b2.c</code>
The matches do not include path separators.	<code>-include-sources "app/*.c"</code> matches: <code>/sources/app/main.c</code>

Glob Pattern Syntax	Example
<p>Pattern starts with slash '/' (UNIX) or drive letter (Windows).</p> <p>Pattern matches absolute path only.</p>	<p>-include-sources "/a" does not match anything.</p> <p>-include-sources "/sources/app" matches:</p> <pre>/sources/app/main.c</pre>
<p>Pattern ends with a slash (UNIX), backslash (Windows), or '**'.</p> <p>Pattern matches all files under specified folder.</p> <p>'**' is ignored if it is at the start of the pattern.</p>	<p>-include-sources "a/" matches</p> <pre>/sources/lib/a/a1.c /sources/lib/a/a2.c</pre>
<p>Pattern contains '**/' (UNIX) or '**\' (Windows). Pattern matches zero or more folders in the specified path.</p>	<p>-include-sources "lib/**/?1.c" matches:</p> <pre>/sources/lib/a/a1.c /sources/lib/b/b1.c</pre>
<p>Pattern starts with '.' or '..'.</p> <p>Pattern matches paths relative to the path where you run the command.</p>	<p>If you start polyspace-configure from /sources/lib/a,</p> <p>-include-sources "../lib/**/b?.c" matches:</p> <pre>/sources/lib/b/b1.c /sources/lib/b/b2.c</pre>
<p>Pattern is a UNC path on Windows .</p>	<p>If your files are on server myServer:</p> <pre>\\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.

Configure Polyspace Analysis Options in User Interface and Generate Scripts

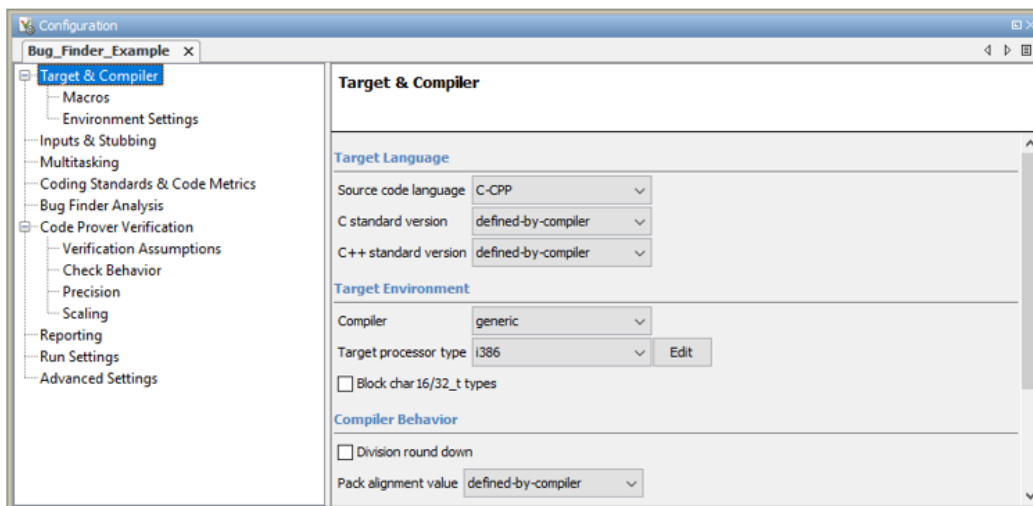
In this section...

“Prerequisites” on page 2-17

“Generate Scripts from Configuration” on page 2-17

“Run Analysis with Generated Scripts” on page 2-18

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:

- Compilation assistant that suggests workarounds for some compilation errors,
- 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\R2021a. Open the Polyspace user interface using the `polyspace` executable and create a new project.

See “Add Source Files for Analysis in Polyspace User Interface” on page 1-2.

- 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 8-2.

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

See “Run Polyspace Analysis on Desktop” on page 1-7.

- 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\R2021a\polyspace\bin\polyspace-code-prover.exe"
```

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

```
"C:\Program Files\Polyspace Server\R2021a\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 2-2

Run Polyspace Analysis with MATLAB Scripts

- “Integrate Polyspace with MATLAB and Simulink” on page 3-2
- “Get Started with Polyspace Analysis by Using MATLAB” on page 3-5
- “Run Polyspace Analysis by Using MATLAB Scripts” on page 3-9
- “Compare Results from Different Polyspace Runs by Using MATLAB Scripts” on page 3-13
- “Generate MATLAB Scripts from Polyspace User Interface” on page 3-16
- “Troubleshoot Polyspace Analysis from MATLAB” on page 3-18

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”.

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

- Your default MATLAB root folder is C:\Program Files\MATLAB\R2021a.
- Your default Polyspace root folder is C:\Program Files\Polyspace\R2021a.

To run Polyspace from within MATLAB, Simulink, or MATLAB Coder™, perform a post-installation step 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.

Integrate Polyspace with MATLAB or Simulink from Same Release

If your Polyspace and MATLAB are both from the same release, integrate them after installation to:

- Run a Polyspace analysis on C/C++ code that is generated from a model or included as custom code in a model from the Simulink Editor. You can also use a MATLAB script to run such analyses.

See “Polyspace 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 “Polyspace Analysis in MATLAB Coder”.

- Run a Polyspace analysis on hand-written C/C++ code by using MATLAB scripts.

See “Polyspace Analysis with MATLAB Scripts”.

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 3-4.

Integrate Polyspace with MATLAB or Simulink

- 1 Open MATLAB with administrator or root privilege.
- 2 At the MATLAB command prompt, enter:

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

If you installed Polyspace in the default folder C:\Program Files\Polyspace\R2021a, 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. You might be prompted that the workspace will be cleared and that all open models will be 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 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.

Integrate Polyspace with MATLAB or Simulink Installation from Earlier Release

In certain cases, it is possible to integrate Polyspace with MATLAB or Simulink from an earlier release. This cross-release integration offers limited functionalities. 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 5-62.

Prerequisite

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

- Your MATLAB or Simulink must support cross-release integration with Polyspace. See “Polyspace Support of MATLAB and Simulink from Different Releases” on page 5-62.
- Your 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 3-4.

Integrate Polyspace with Cross-Release MATLAB or Simulink

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

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

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

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

- 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
```

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

The integration of MATLAB and Polyspace adds some 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 5-62
- “Polyspace Analysis with MATLAB Scripts”
- “Polyspace Analysis in Simulink”
- “Polyspace Analysis in MATLAB Coder”

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 5-15.

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 3-2.

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>clearv</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 3-9
- “Visualize Bug Finder Analysis Results in MATLAB” on page 19-9
- “Troubleshoot Polyspace Analysis from MATLAB” on page 3-18
- “Generate MATLAB Scripts from Polyspace User Interface” on page 3-16

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 3-2.

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 3-18.

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');
```

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`

Related Examples

- “Generate MATLAB Scripts from Polyspace User Interface” on page 3-16
- “Visualize Bug Finder Analysis Results in MATLAB” on page 19-9
- “Troubleshoot Polyspace Analysis from MATLAB” on page 3-18

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 = datetimestr('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

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

More About

- “Run Polyspace Analysis by Using MATLAB Scripts” on page 3-9

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, compilation assistant, 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 3-2.

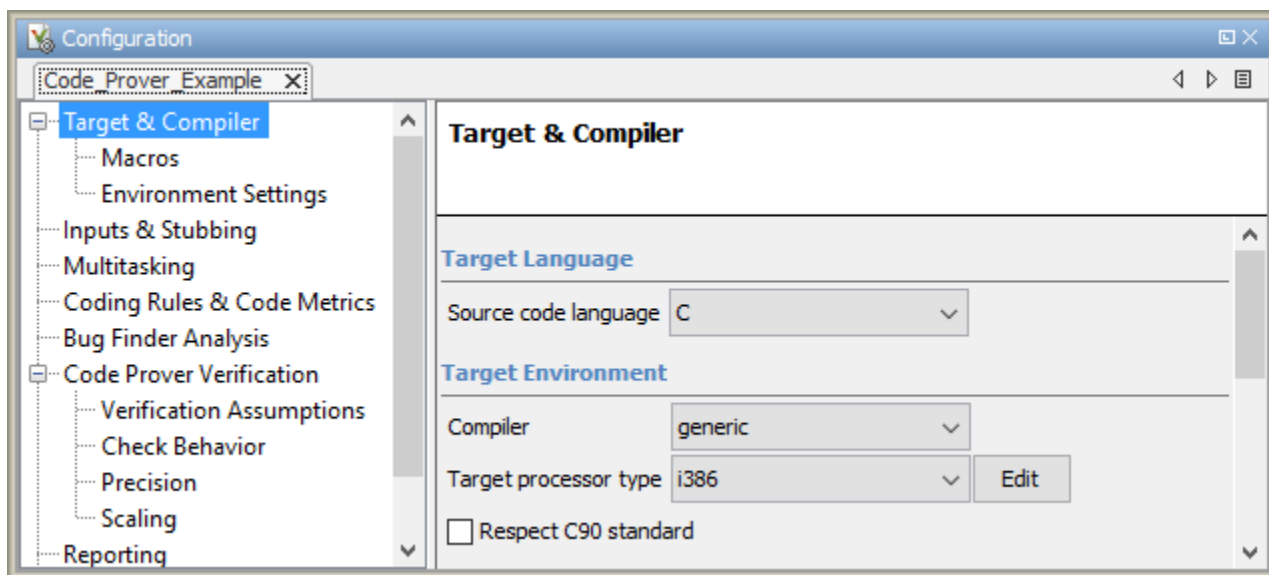
Create Scripts from Polyspace Projects

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 User Interface” on page 1-2.

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 3-9

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 3-2.

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');

```

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 3-9
- “Troubleshoot Compilation Errors”

Offload Polyspace Analysis to Remote Servers from Desktop

- “Send Polyspace Analysis from Desktop to Remote Servers” on page 4-2
- “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts” on page 4-6

Send Polyspace Analysis from Desktop to Remote Servers

In this section...

“Client-Server Workflow for Running Analysis” on page 4-2

“Prerequisites” on page 4-3

“Offload Analysis in Polyspace User Interface” on page 4-3
--

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 4-6.
- 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.

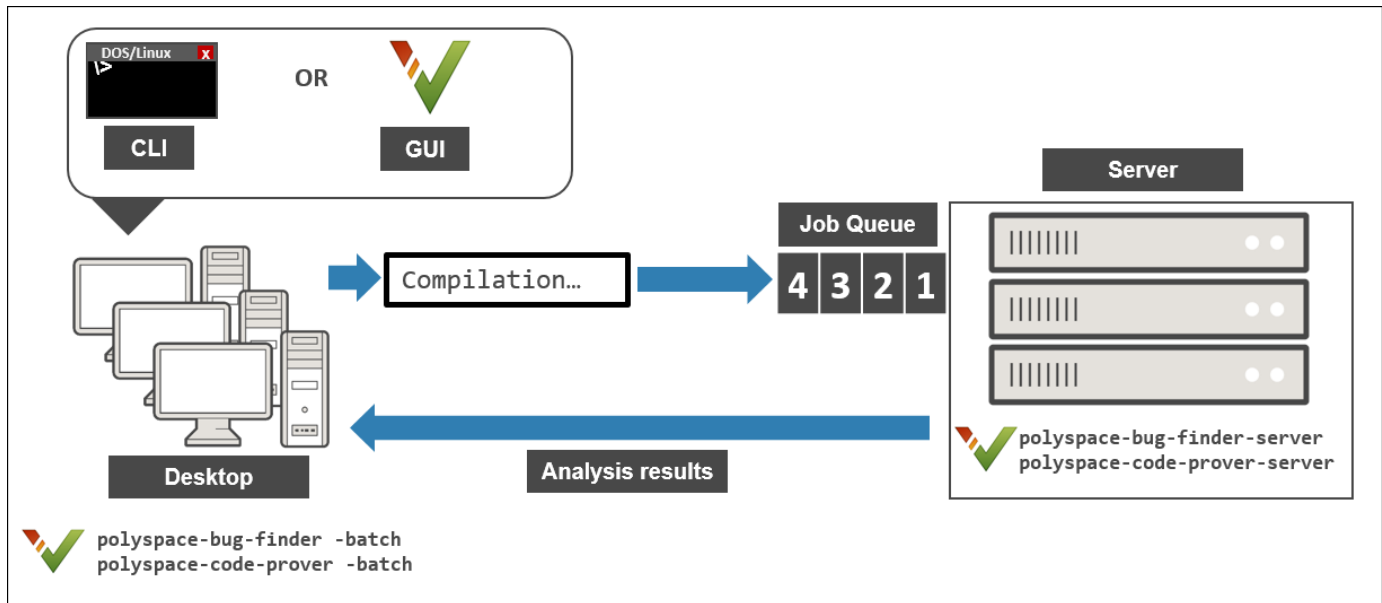
You require the Polyspace desktop product, Polyspace Bug Finder on the computer that acts as the client node.

- 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.



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 User Interface” on page 1-2.
- How to set up communication between client and server, see “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

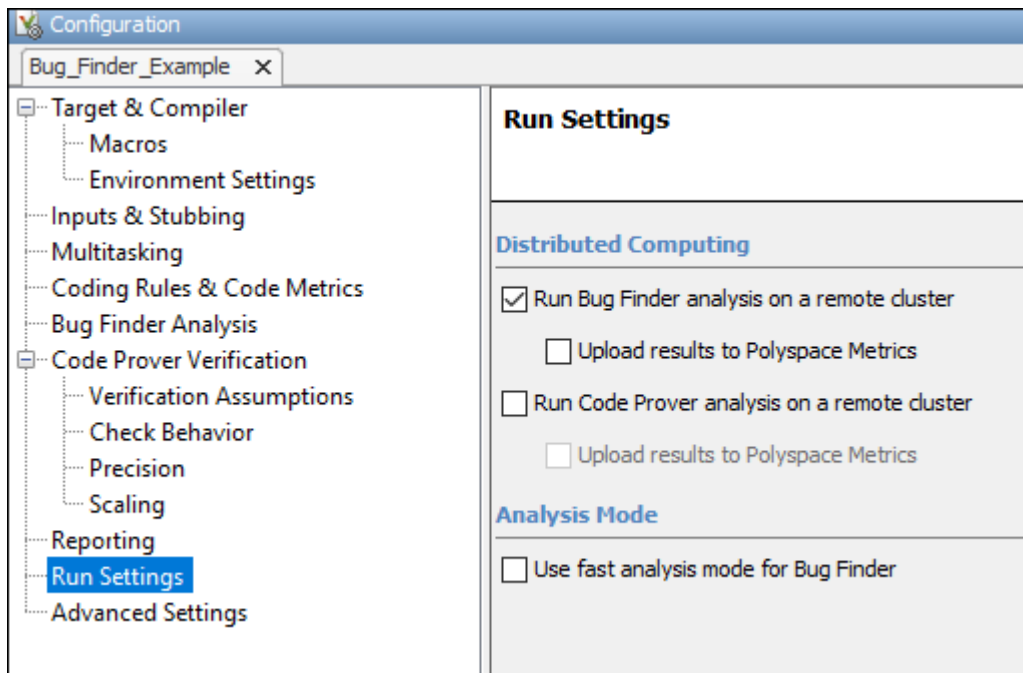
Once you have set up a Polyspace project and established communication between a desktop and a remote server, you are ready to offload a Polyspace analysis.

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 If you want to store your results in the Polyspace Metrics repository, select **Upload results to Polyspace Metrics**.

Otherwise, clear this check box. After analysis, the results are downloaded to the desktop for review.

- 4 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.

- 5 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 you uploaded the results to Polyspace Metrics, you have to explicitly download them from the Polyspace Metrics 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`.

Note If you choose to upload results to Polyspace Metrics, your results are not downloaded automatically after verification. Use the Polyspace Metrics web dashboard to view the results and download them to your desktop. For more information, see “View Projects in Polyspace Metrics” on page 20-4.

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 4-6

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 4-2.
- 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.

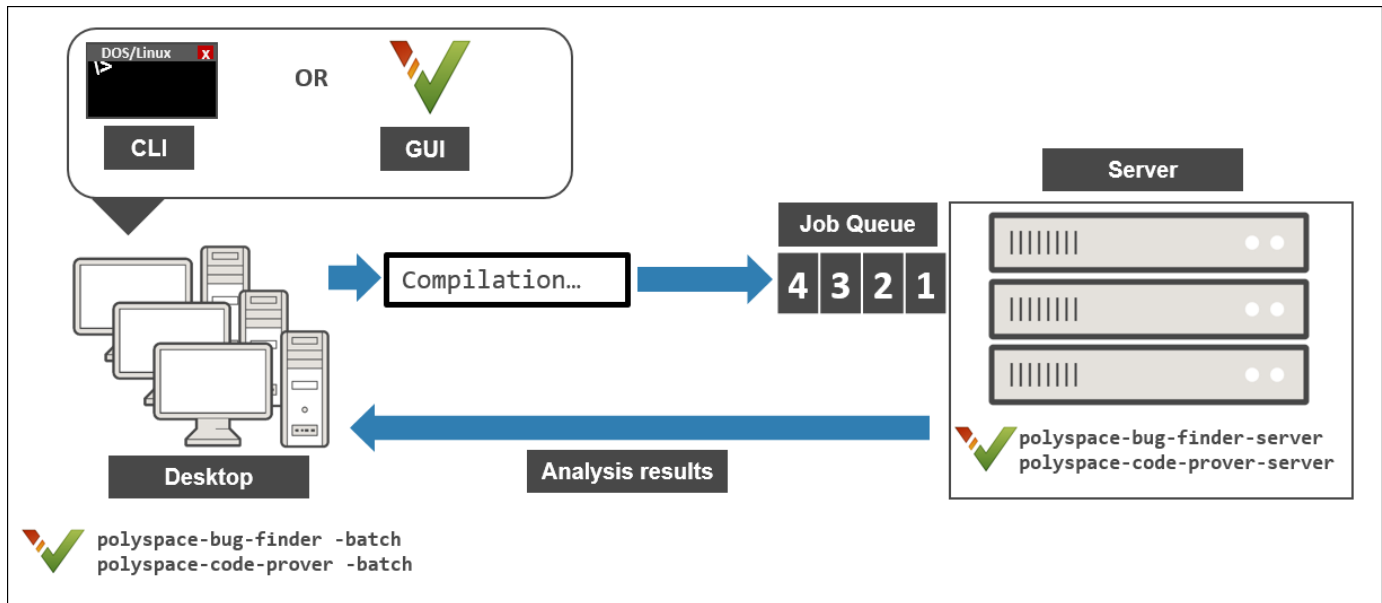
You require the Polyspace desktop product, Polyspace Bug Finder on the computer that acts as the client node.

- 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.



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 this command:

```
polyspaceroot\polyspace\bin\polyspace-bug-finder
  -batch -scheduler NodeHost|MJSName@NodeHost [options] [-mjs-username name]
```

where:

- *polyspaceroot* is the installation folder of Polyspace desktop products, for instance, C:\Program Files\Polyspace\R2021a.
- *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 **Metrics > Metrics and Remote Server Settings**. Open the MATLAB Parallel Server Admin Center. Under **MATLAB Job Scheduler**, see the **Name** and **Hostname** columns 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 “Analysis Options in Polyspace Bug Finder”. 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 2-16.
- 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).

The analysis executes locally on your desktop up to the end of the compilation phase. 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` in the results folder. To monitor your analysis, use the `polyspace-jobs-manager` command with the job ID.

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*.

When the analysis job is queued on the server, the command `polyspace-bug-finder` returns a job id. In addition, a file `ID.txt` in 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 localhost
    -results-dir "%RESULTS_PATH%" -options-file "%OPTIONS_FILE%"
pause
```

polyspaceroot is the Polyspace installation folder. *localhost* 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 4-2

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.

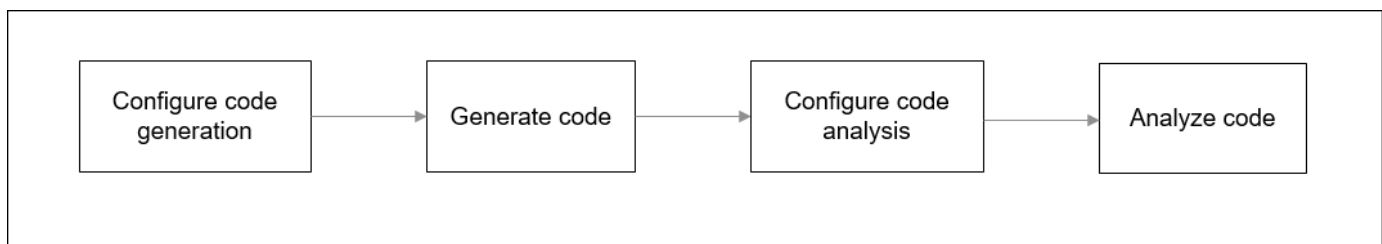
This topic uses Embedded Coder for code generation. For analysis of TargetLink-generated code, see “Run Polyspace Analysis on Code Generated with TargetLink” on page 5-56.

For a tutorial with a specific model, see “Run Polyspace Analysis on Code Generated from Simulink Model” on page 5-15.

Prerequisites

Before you run Polyspace from Simulink, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 3-2.

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 5-43.

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.

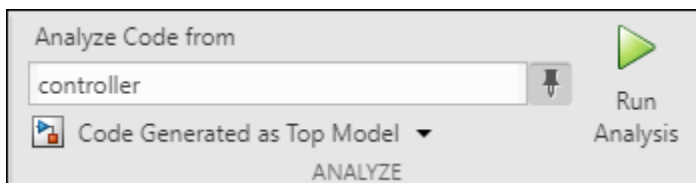
Configure Code Analysis

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

- 1 Select the product to run: **Bug Finder** or **Code Prover**.
- 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 bugs. 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, Polyspace first launches the code generation process and then starts the analysis.

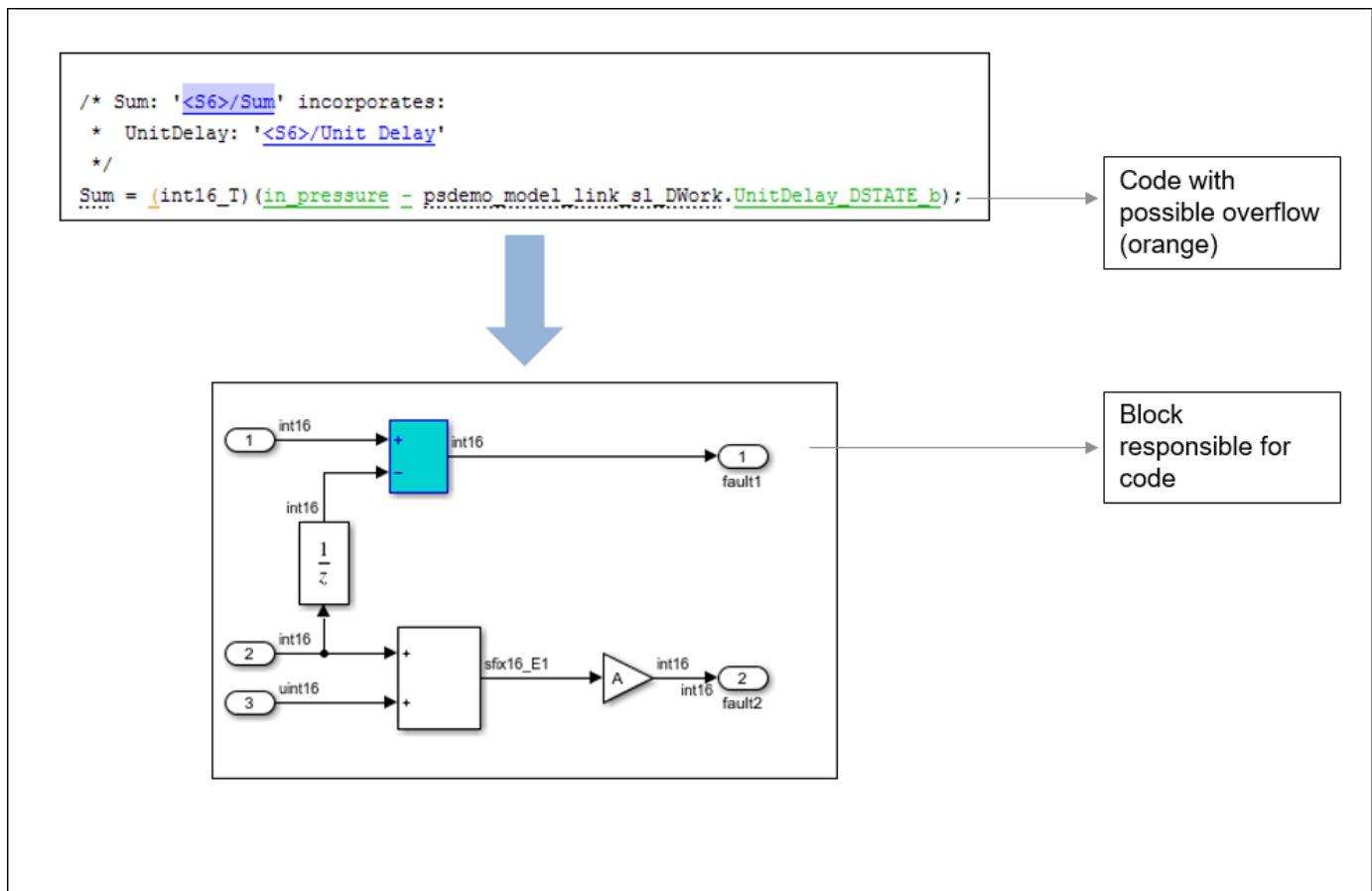
If the current model is referenced in another model and you want to verify the generated code in the context where the model is referenced, instead of **Code Generated as Top Model**, use **Code Generated as Model Reference**. In the latter case, Polyspace does not launch code generation automatically 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 16-2
-
- “Address Polyspace Results Through Bug Fixes or Justifications” on page 17-2
- “Filter and Group Results in Polyspace Desktop User Interface” on page 18-2

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 5-60.

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.

If you include handwritten C/C++ code in S-function blocks, the Polyspace analysis can reveal possible integration issues between the handwritten and generated code. You can also analyze the handwritten code in isolation. See “Run Polyspace Analysis on S-Function Code” on page 5-27.

Annotate Blocks to Justify Issues

If you do not want to make changes to the model in response to a Polyspace result, annotate the relevant blocks. After you annotate a block, code operations generated from the block show results that are prepopulated 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.

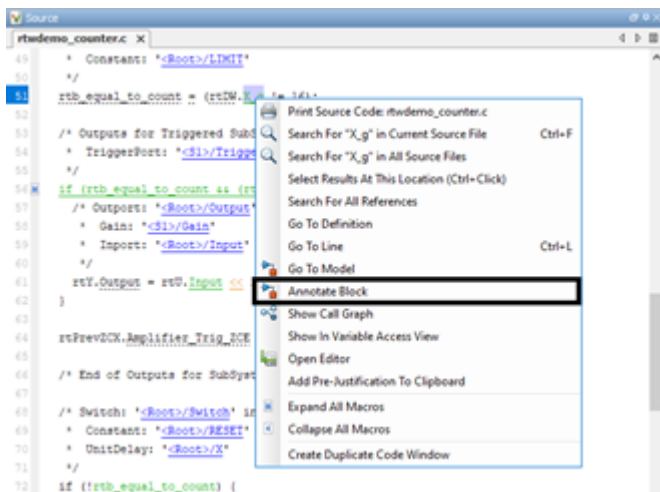
Annotations in Simulink blocks or in generated code do not take the history of the analysis into account. If you update your model, the Polyspace results might change while the annotations do not. Updating the model might render the existing annotations outdated. Check your annotations when you update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

- 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 and from the context menu, select **Annotate Block**.



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 User Interface.

Polyspace uses the user-provided information to prepopulate the annotations in Simulink. Comments that are set in the Polyspace User Interface appear within double quotes in the **Comment** field in Simulink. If you have double quotes in the comment in Polyspace User interface, those are replaced by single quotes in Simulink.

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 menu **Annotation Type**.
- If you want to annotate multiple results of the same type, enter a comma separated list of result acronym in the text box. See:
 - “Short Names of Bug Finder Defect Checkers” on page 14-26
 - “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 menu. Select the result that you want to annotate from this dropdown menu.
- 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 User Interface.

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).

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

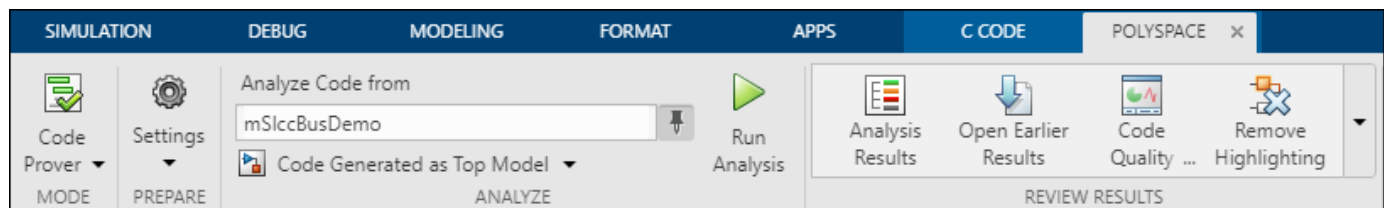
- “Configure Advanced Polyspace Options in Simulink” on page 5-45

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_lprj` 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
Open remote job monitor (if you are offloading the analysis to a server).	<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>
Open Polyspace Metrics or Polyspace Access web interface if you are using one of them to host Polyspace results.	<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.</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 5-2

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 5-62.

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 5-15.

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 3-2.

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 5-43.

```
% 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 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';
```

- 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 “Analysis Options in Polyspace Bug Finder”.

- 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);
```

Alternatively, upload the results to Polyspace Access. See “Upload Results to Polyspace Access” (Polyspace Bug Finder Access).

- 2 Address the results. For more information, see “Address Polyspace Results Through Bug Fixes or Justifications” on page 17-2.

See Also

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

More About

- “Run Polyspace Analysis on Generated Code by Using Packaged Options Files” on page 5-21
- “Run Polyspace Analysis by Using MATLAB Scripts” on page 3-9
- “Polyspace Support of MATLAB and Simulink from Different Releases” on page 5-62

Run Polyspace Analysis on Code Generated from Simulink Model

You can run Polyspace on the code generated from a Simulink model or subsystem.

- Polyspace Bug Finder checks the code for bugs or coding rule violations (for instance, MISRA C: 2012 rules).
- Polyspace Code Prover exhaustively checks the code for run-time errors.

If you use Embedded Coder for code generation, this tutorial shows how to run Polyspace on the code generated from a simple Simulink model. For the full workflow, see “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 5-2.

Prerequisites

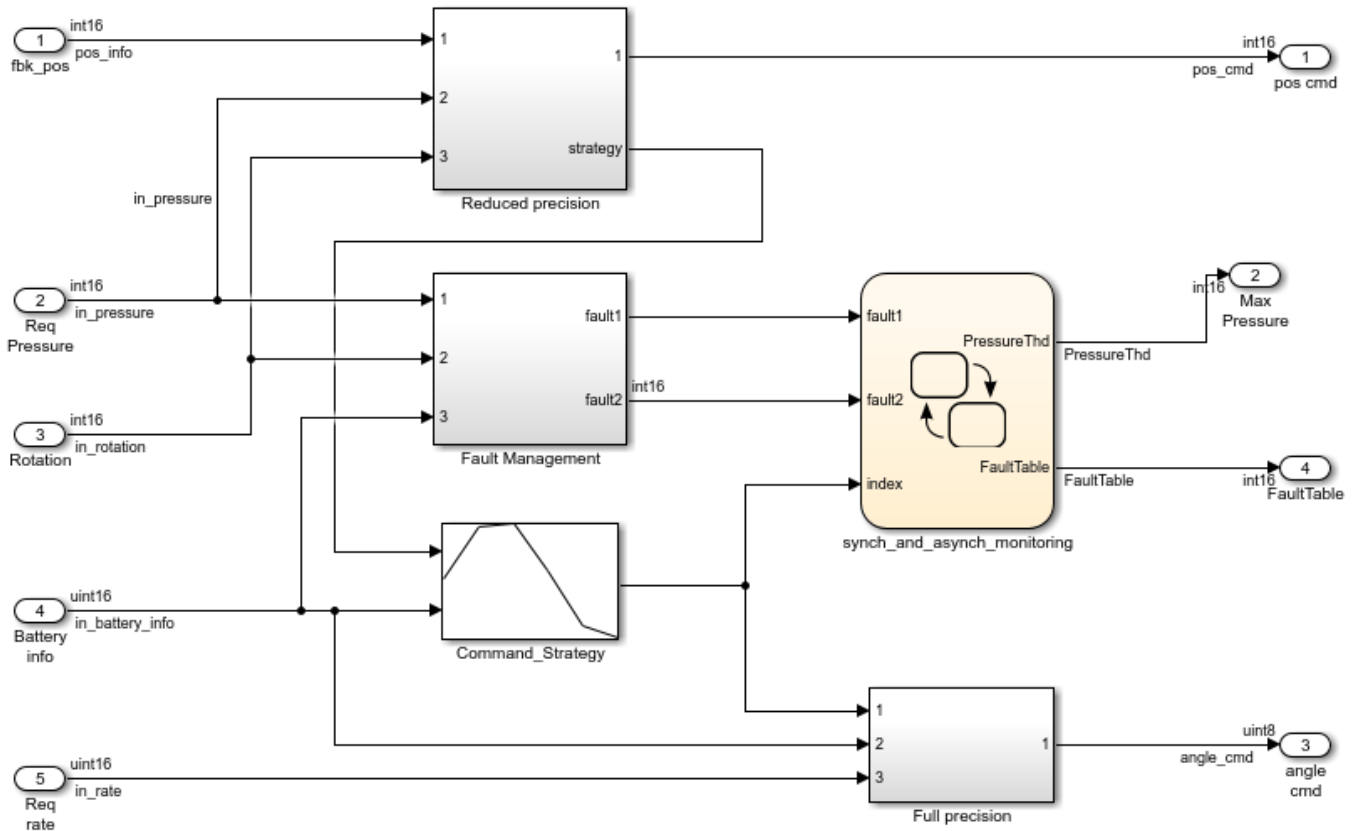
Before you run Polyspace from Simulink, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 3-2.

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

```
openExample('polyspace_code_prover/OpenModelForCodeGenerationAndPolyspaceAnalysisExample')
```

Open Model for Code Generation and Polyspace Analysis

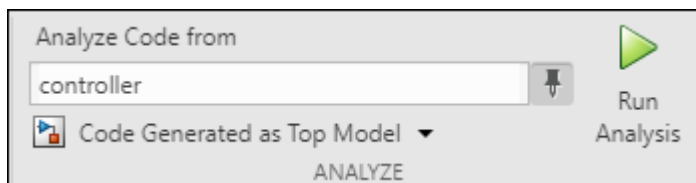
Open the model `polyspace_controller_demo` for configuring code generation and Polyspace analysis.



Generate and Analyze Code

To start the Polyspace analysis of code generated from the model:

- 1 On the **Apps** tab, select **Polyspace Code Verifier**.
- 2 On the **Polyspace** tab, locate the **Analyze** field and select **Code Generated as Top model** from the drop-down menu.
- 3 Click anywhere on the canvas. The **Analyze Code from** field shows the model name. Select **Run Analysis**.



Polyspace checks for generated code when you click **Run Analysis**. If no generated code is present, Polyspace first generates the code by using Embedded Coder and then starts the analysis.

Equivalent MATLAB Code:

```
load_system('polyspace_controller_demo');
slbuild('polyspace_controller_demo');
```

```
mlopts = pslinkoptions('polyspace_controller_demo');
mlopts.ResultDir = '\result';
mlopts.VerificationMode = 'CodeProver';
pslinkrun('polyspace_controller_demo', mlopts);
```

To analyze with Bug Finder, replace CodeProver with BugFinder. For more information on the code, see pslinkoptions and pslinkrun.

Review Analysis Results

After analysis, the results are displayed in the Polyspace user interface.

If you run Bug Finder, the results consist of bugs detected in the generated code. If you run Code Prover, the results consist of checks that are color-coded as follows:

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

Review each analysis result in detail. For instance, in your Code Prover results:

- 1 On the **Results List** pane, select the red **Out of bounds array index** check.
- 2 On the **Source** pane, place your cursor on the red check to view additional information. For instance, the tooltip on the red [operator states the array size and possible values of the array index. The **Result Details** pane also provides this information.

The error occurs in a handwritten C file Command_strategy_file.c. The C file is inside an S-function block Command_Strategy in the Reduced Precision subsystem (in a model reference relative threshold).

Trace Errors Back to Model and Fix Them

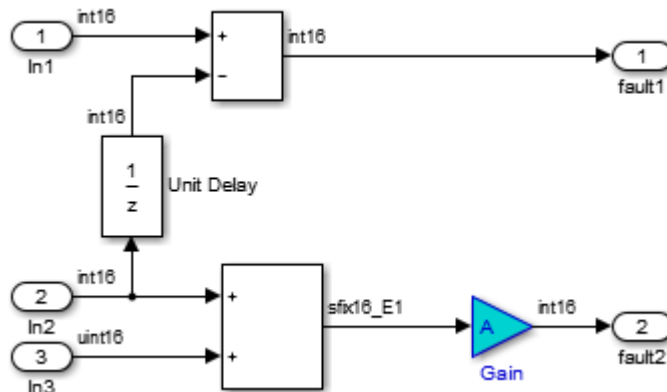
For code generated from the model, you can trace an error back to your model. These sections show how to trace specific Code Prover results back to the model.

Error 1: Out of bounds array index

- 1 On the **Results List** pane, select the orange **Out of bounds array index** error that occurs in the file polyspace_controller_demo.c.
- 2 On the **Source** pane, click the link **S4:76** in comments above the orange error.

```
/* Transition: '<S4>:75' */
/* Transition: '<S4>:76' */
(*i)++;

/* Output: '<Root>/FaultTable' */
polyspace_controller_demo_Y.FaultTable[*i] = 10;
```

You can avoid the **Overflow** in several ways. One way is to constrain the value of the signal `in_battery_info` that is fed to the Sum block. To constrain the signal:

- 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.

The errors in this model occur due to one of the following:

- Faulty scaling, unknown calibrations and untested data ranges coming out of a subsystem into an arithmetic block.
- Array manipulation in Stateflow event-based modelling and handwritten lookup table functions.
- Saturations leading to unexpected data flow inside the generated code.
- Faulty Stateflow programming.

Once you identify the root cause of the error, you can modify the model appropriately to fix the issue.

Check for Coding Rule Violations

To check for coding rule violations, before starting code analysis:

- 1 On the **Polyspace** tab, select **Settings**.
- 2 In the Configuration Parameters dialog box, select an appropriate option in the **Settings from** list. For instance, select `Project configuration` and `MISRA C 2012 AGC Checking`.

It is recommended that you run Bug Finder for checking MISRA C:2012 rules. On the **Polyspace** tab, select **Bug Finder**.

- 3 Click **Apply** or **OK** and rerun the analysis.

Annotate Blocks to Justify Results

You can justify your results by adding annotations to your blocks. During code analysis, Polyspace Code Prover reads your annotations and populates the result with your justification. Once you justify a result, you do not have to review it again.

- 1 On the **Results List** pane, from the drop-down list in the upper left corner, select **File**.

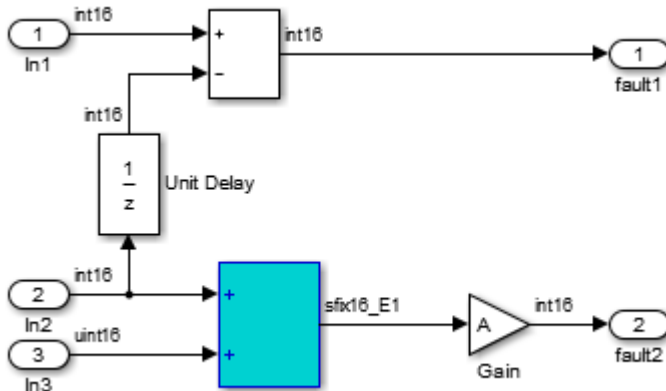
- 2 In the file `polyspace_controller_demo.c`, in the function `polyspace_controller_demo_step()`, select the violation of MISRA C:2012 rule 10.4. The **Source** pane shows that an addition operation violates the rule.
- 3 On the **Source** pane, click the link **S1/Sum1** in comments above the addition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

```

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

```

You see that the rule violation occurs in a Sum block.



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.
- d Regenerate code and rerun the analysis. The **Severity** and **Status** columns on the **Results List** pane are repopulated with your annotations.

See Also

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 5-2

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 without access to 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 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.
 - `linkdata.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 `linkdata.xml` is required. In case the file `linkdata.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 `linkdata.xml` in the same folder as your Polyspace results. You cannot link the code with Simulink model if you do not have the file `linkdata.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

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

More About

- “Analyze Code Generated as Standalone Code in a Distributed Workflow” (Simulink)
- “Run Polyspace Analysis by Using MATLAB Scripts” on page 3-9

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:

- A custom code analysis detects issues in the custom code that might cause run-time errors or bugs in the simulation. This analysis uses target configuration that is compatible with Simulink simulation.
- A generated code analysis detects bugs, run-time errors, and inefficiencies in the code. generated from the complete model This analysis uses the settings that you specify in **Hardware Implementation** pane of the Configuration Parameters dialog box to configure the Polyspace **Target processor type** settings.

Prerequisite

Before you run Polyspace with Simulink, link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 3-2.

Analyze Custom Code

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

- C Function: See “Call and Integrate External C Algorithms into Simulink” (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 5-27.
- 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 5-29.
- 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 5-37.

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 16-2
-
-
- “Address Polyspace Results Through Bug Fixes or Justifications” on page 17-2
- “Filter and Group Results in Polyspace Desktop User Interface” on page 18-2

To fix the flagged issues, modify the code. For more information, see “Fix Identified Issues” on page 5-41. Alternatively, modify the Simulink model to resolve the Polyspace results. See “Fix Issues” on page 5-30.

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 5-19.

See Also

`pslinkoptions` | `pslinkrun`

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 5-2
- “Run Polyspace Analysis on S-Function Code” on page 5-27
- “Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts” on page 5-29

- “Run Polyspace Analysis on Custom Code in C Function Block” on page 5-37
- “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 3-2.

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 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 18-2.
 - For help reviewing and understanding the results, see “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 `-sl dv`.

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**.
- Select **Settings**. In the Configuration Parameters dialog box, make sure that the following parameters are set:
 - **Settings from** — Project configuration and MISRA C 2012 checking
 - **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 5-2
- “Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts” on page 5-29

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 3-2.

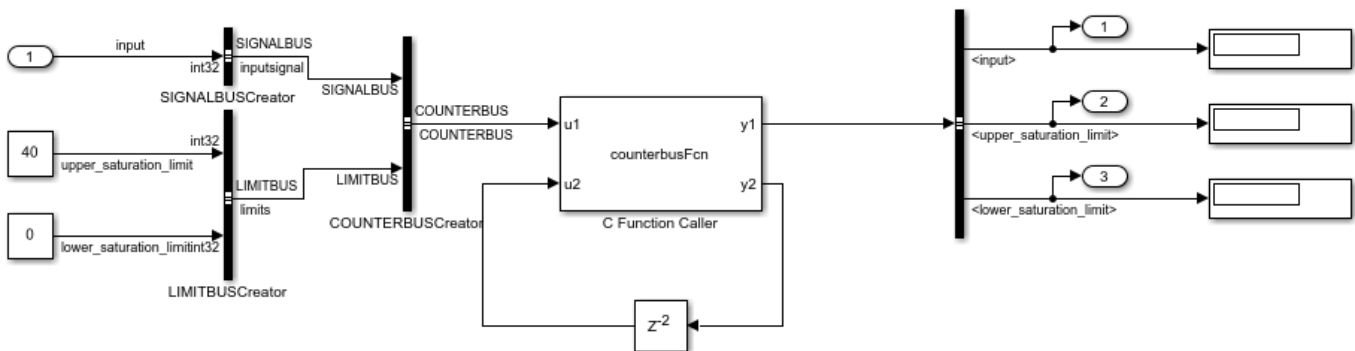
To open the models used in this example, look for this example in the MATLAB Help browser and click the **Open Model** buttons.

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 `mSlcCbBusDemo` 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

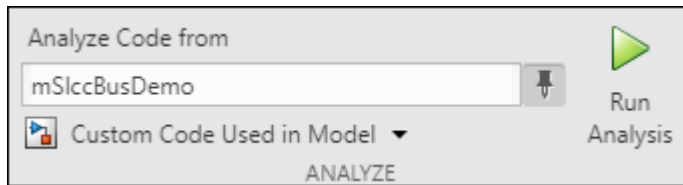
Configure analysis options and run Polyspace.

- 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**.

- 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:

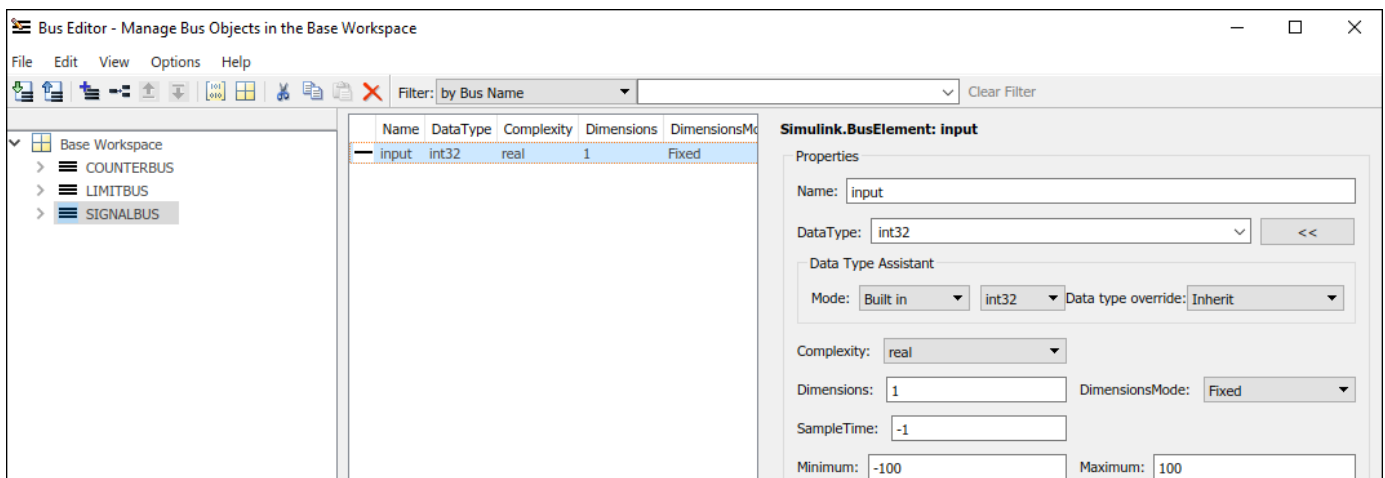
```
limit = u1->inputsignal.input + u2;
```

The operands come from inputs to counterbusFcn, which in turn come from these inputs to the C Caller block:

- The bus signal 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 input in several ways. For instance, you can constrain the bus signal variable SIGNALBUS that comes from input:

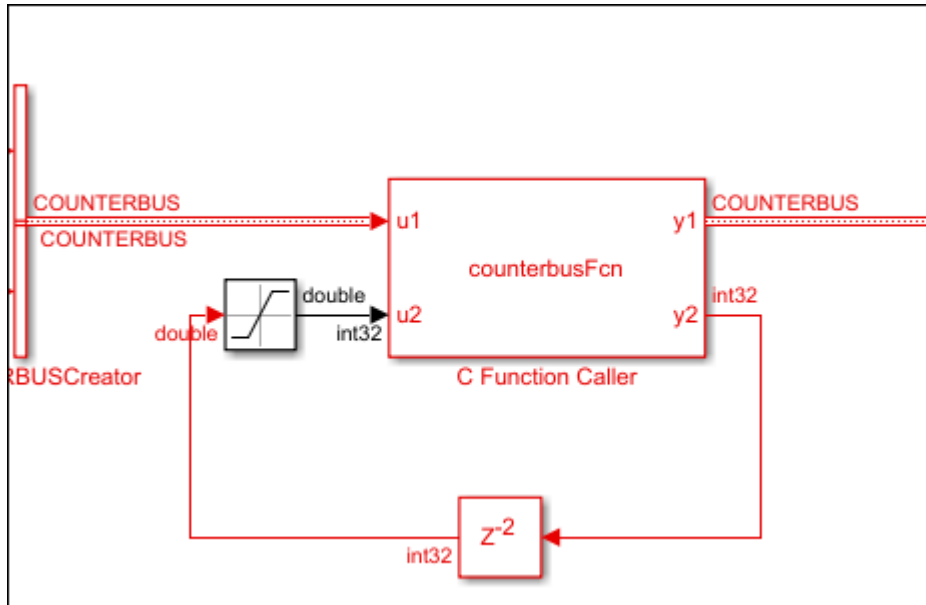
- 1 In the Simulink Editor, open the Model Explorer from the **Modeling** tab.
- 2 The base workspace variables contain the variable SIGNALBUS. Select this variable and open the bus editor to edit this variable. Specify a minimum and maximum value for the variable.



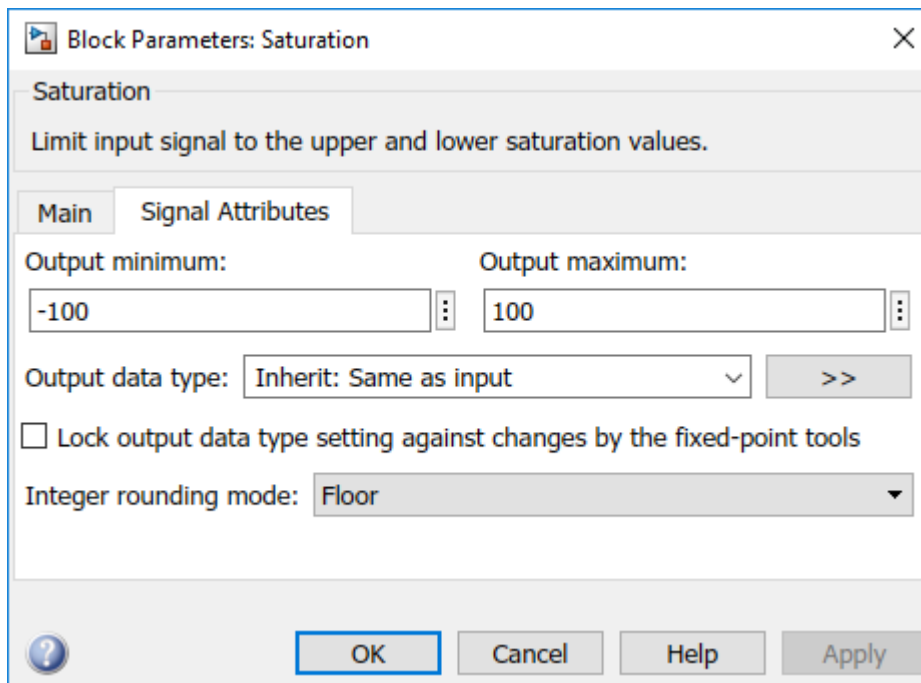
Save the bus object in a MAT-file. You can overwrite the file dLctData.mat or create a file.

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

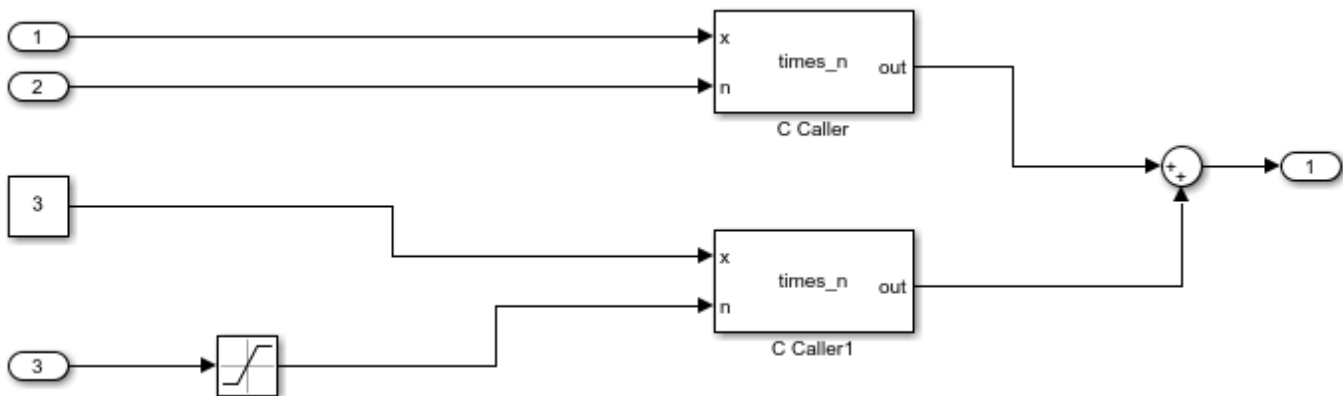
This example uses a function called from multiple C Caller blocks in the model. The function simply returns the product of its two arguments.

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');
```



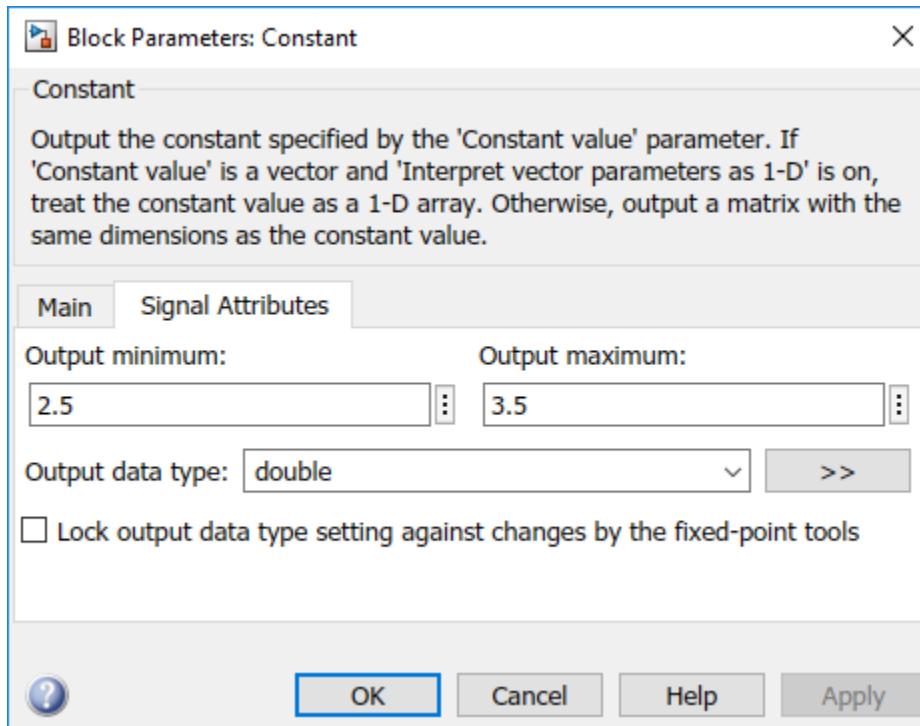
Inspect Model

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:

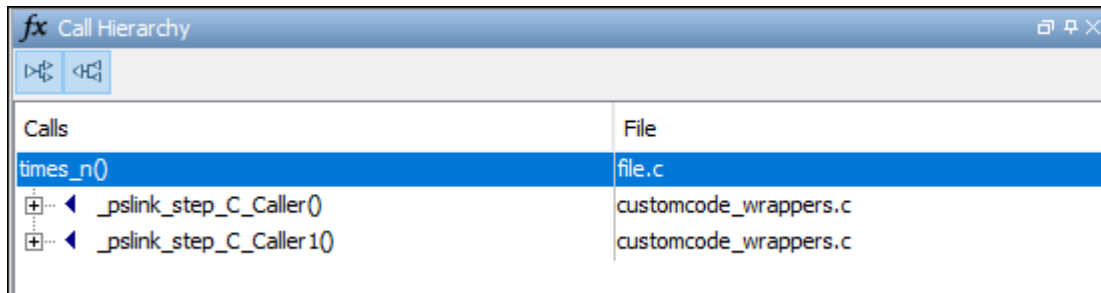
- The overflow occurs in both call contexts.
- The overflow is proven to not occur in one context (green check) and might occur in the other context (orange check).

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
<code>times_n()</code>	<code>file.c</code>
<code>_pslink_step_C_Caller()</code>	<code>customcode_wrappers.c</code>
<code>_pslink_step_C_Caller1()</code>	<code>customcode_wrappers.c</code>

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 5-2
- “Run Polyspace Analysis on S-Function Code” on page 5-27

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

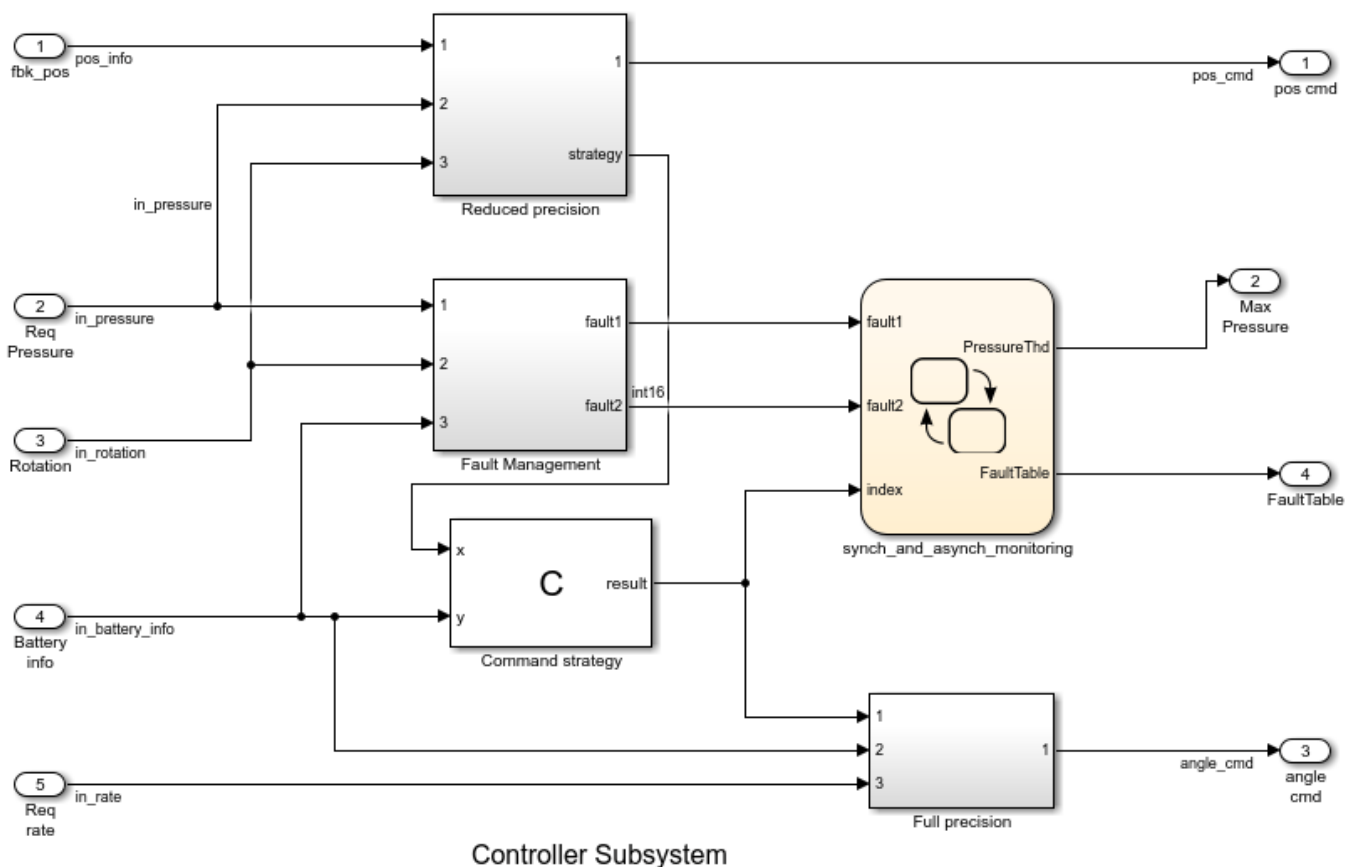
Before you run Polyspace from Simulink, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 3-2.

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.



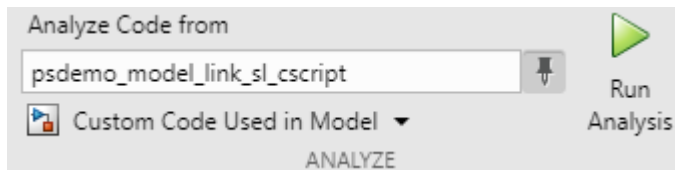
The **Command Strategy** block implements a look-up table using custom C code and outputs a value result based on two inputs *x* and *y*.


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** from the drop-down list located at the leftmost corner of the **Polyspace** tab.
- 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. Select **Bug Finder** or **Code Prover** from the drop-down list located at the leftmost corner of the **Polyspace** tab and rerun the analysis.

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” on page 16-25 and “Source” on page 16-18.

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 16-2.

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 < 92))
//...
```

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 5-29.

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 Polyspace Results Through Bug Fixes or Justifications” on page 17-2.

See Also

`pslinkoptions` | `pslinkrun`

More About

- “Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts” on page 5-29
- “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>IncludeHyperlinksInReport</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 Advanced Polyspace Options in Simulink

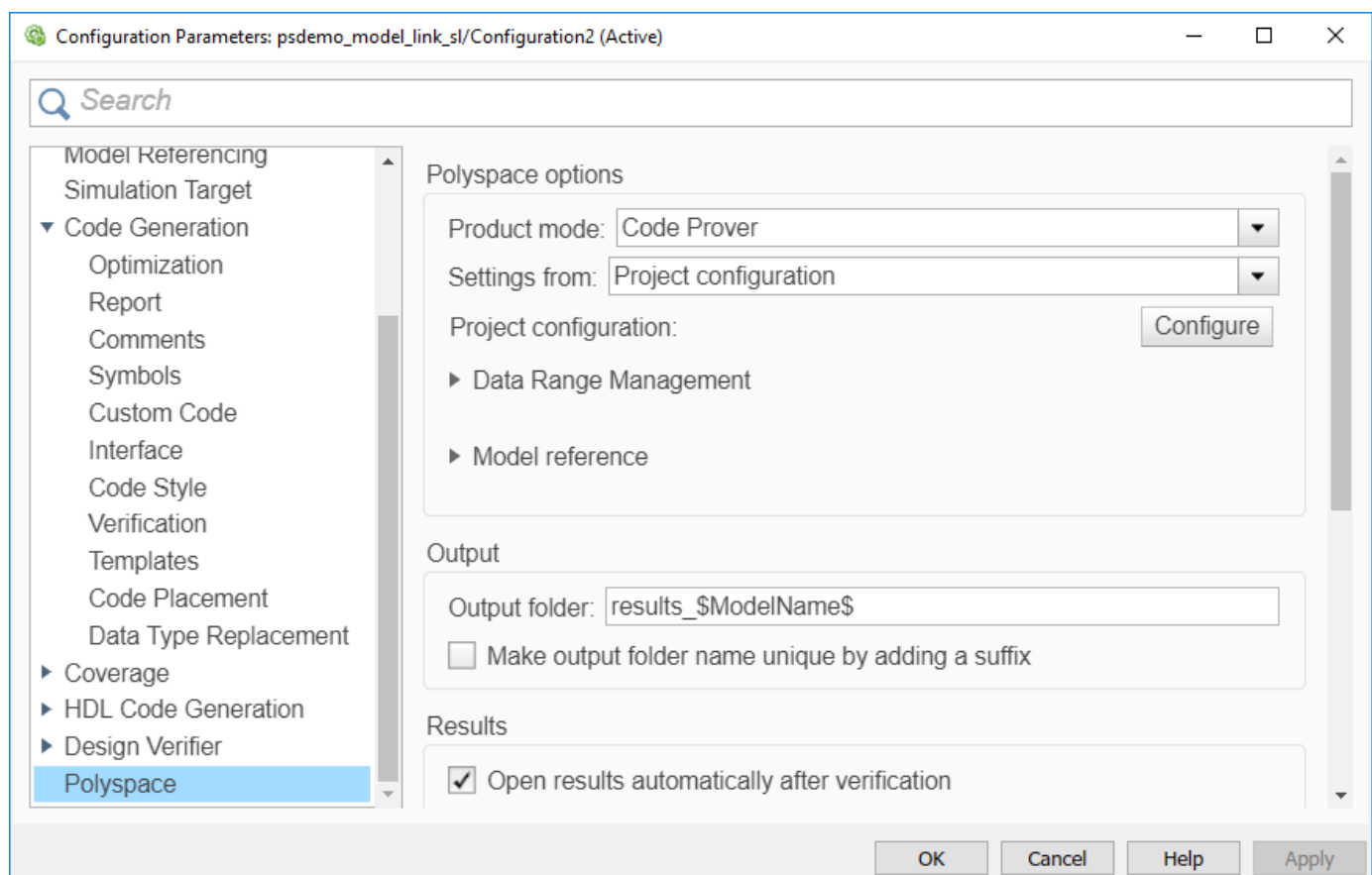
Before analyzing generated code in Simulink, you can change some of the default options. This topic shows how to configure the options and save this configuration.

For getting started with Polyspace analysis in Simulink, see “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 5-2.

Configure Options

Set basic options

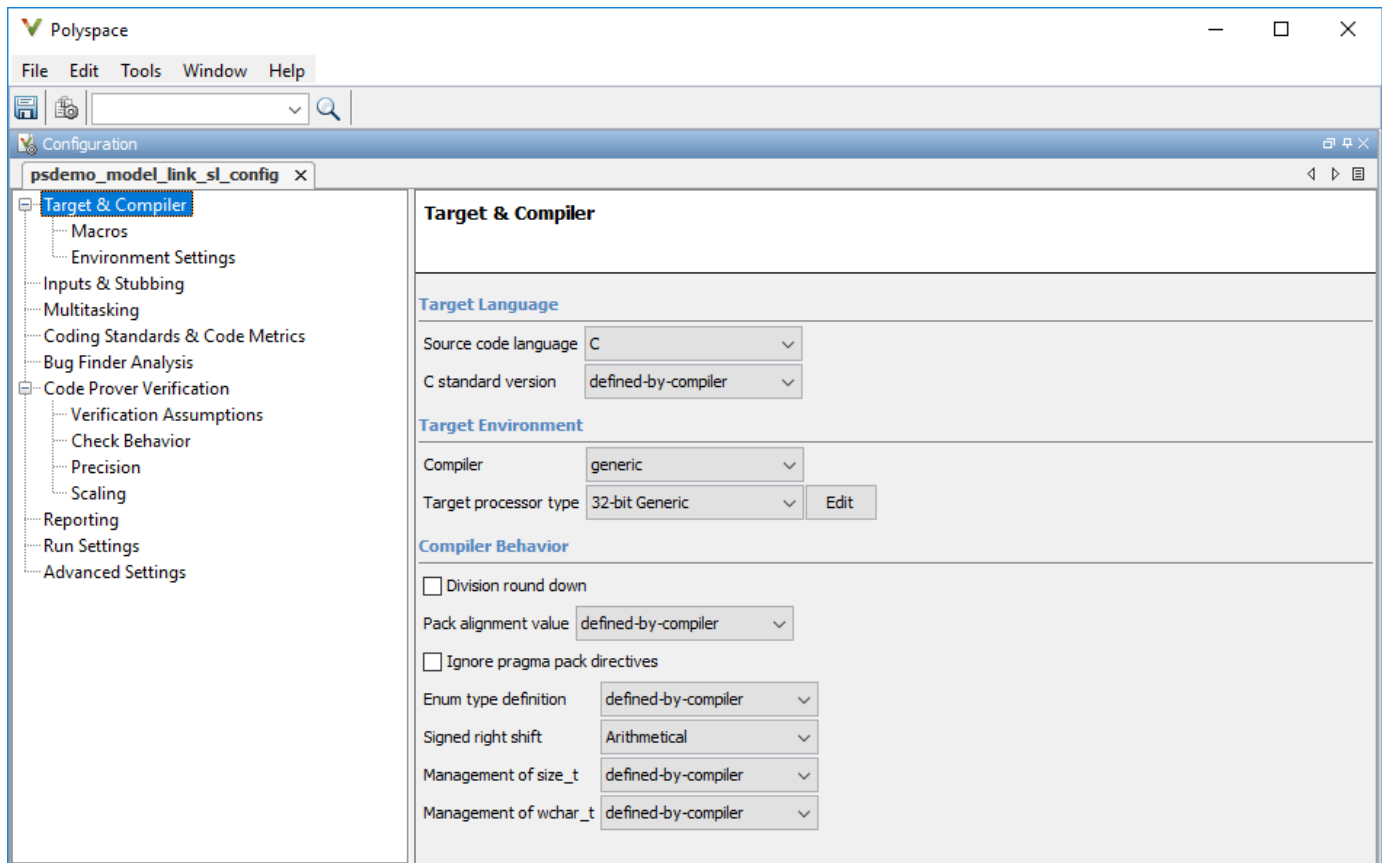
The commonly used options appear in Simulink Configuration Parameters.



To open the 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 more 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 such as:

- Run the code analysis on a remote cluster. Use the option **Run Bug Finder** or **Code Prover analysis on a remote cluster**.

If you use this option, after starting the analysis, you can follow the progress of the analysis on the remote cluster through the Job Monitor window. On the **Polyspace** tab, select **Remote Job Monitor**.

- Stub certain functions for the analysis and then constrain the function output. Use the options **Functions to stub (-functions-to-stub)** and **Constraint setup (-data-range-specifications)**.

If a basic option in the Configuration Parameters window directly conflicts with an advanced option in the Polyspace window, the former prevails. For instance, in this situation, Polyspace checks for MISRA C: 2012 rules:

- “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.

By default, the advanced options are saved in a project file (*modelname_config.psprj*) in the `pslink_config` subfolder of the results folder. You can reuse the options associated with this project.


Share and Reuse Configuration

You can share the basic or advanced options across multiple models.

- **Basic options:** You can share and reuse the options set in the Configuration Parameters window. See “Share a Configuration with Multiple Models” (Simulink).
- **Advanced options:** The advanced options are saved in a separate Polyspace project associated with your analysis. Share this project across multiple models.

The next sections show how to reuse the advanced options. You can specify the advanced options just once. You can reuse these advanced options across multiple models and set only the basic options individually in each model.

Set options from model

Set the advanced options as needed. To see where the associated project file is stored or change the name of the file, on the Polyspace window toolbar, click the  icon.

Reuse options in another model

To reuse the advanced options in another model, open the Configuration Parameters window from the other model. On the **Polyspace** tab, select **Settings**.

- Select **Use custom project file**. Provide the path to the project file previously created (extension `.psprj`).
- For **Settings from**, select `Project configuration` so that the settings in your project are used.

If you want to check for additional issues, for instance MISRA C: 2012 violations, select `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 5-2
- “Run Polyspace Analysis on Code Generated with TargetLink” on page 5-56
- “Default Polyspace Options for Code Generated with Embedded Coder” on page 5-50
- “Default Polyspace Options for Code Generated with TargetLink” on page 5-58

How Polyspace Analysis of Generated Code Works

When you run Polyspace on generated code, the software automatically reads the following information from the generated code:

- `initialize()` functions
- `terminate()` functions
- `step()` functions
- List of parameter variables
- List of input variables

If you run Code Prover, the software uses this information to generate a `main` function that:

- 1 Initializes parameters using the Polyspace option `Parameters (-variables-written-before-loop)`.
- 2 Calls initialization functions 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 using the option `Step functions (-functions-called-in-loop)`.
- 5 Calls the `terminate` function using the option `Termination functions (-functions-called-after-loop)`.

The `main` function conceptually looks like this:

```
init_parameters    \\ -variables-written-before-loop
init_fct()        \\ -functions-called-before-loop
while(1){         \\ start main loop
  init_inputs      \\ -variables-written-in-loop
  step_fct()       \\ -functions-called-in-loop
}
terminate_fct()   \\ -functions-called-after-loop
```

Code Prover uses this generated `main` function to perform the subsequent analysis.

For C++ code that is 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 5-50

“Constraint Specification” on page 5-50

“Recommended Polyspace options for Verifying Generated Code” on page 5-51

“Hardware Mapping Between Simulink and Polyspace” on page 5-51

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” on page 10-2. 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 5-53.

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 Advanced Polyspace Options in Simulink” on page 5-45.

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.

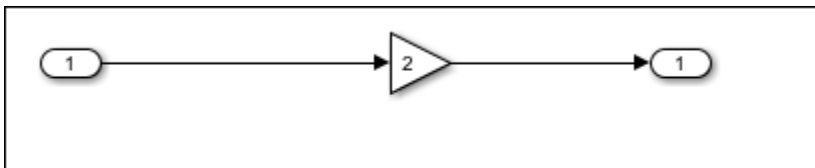
External Constraints on Polyspace Analysis of Generated Code

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.

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

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

See Also

More About

- “Default Polyspace Options for Code Generated with Embedded Coder” on page 5-50
- “Choose Storage Class for Controlling Data Representation in Generated Code” (Embedded Coder)

Run Polyspace Analysis on Code Generated with TargetLink

You can analyze code generated from Simulink models with TargetLink. The versions of TargetLink currently supported are 4.2 and 4.3.

You have fewer capabilities for code generated with TargetLink compared to code generated with Embedded Coder. For instance, you cannot add annotations to your blocks that carry over to the generated code and justify known issues.

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**.
- 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”

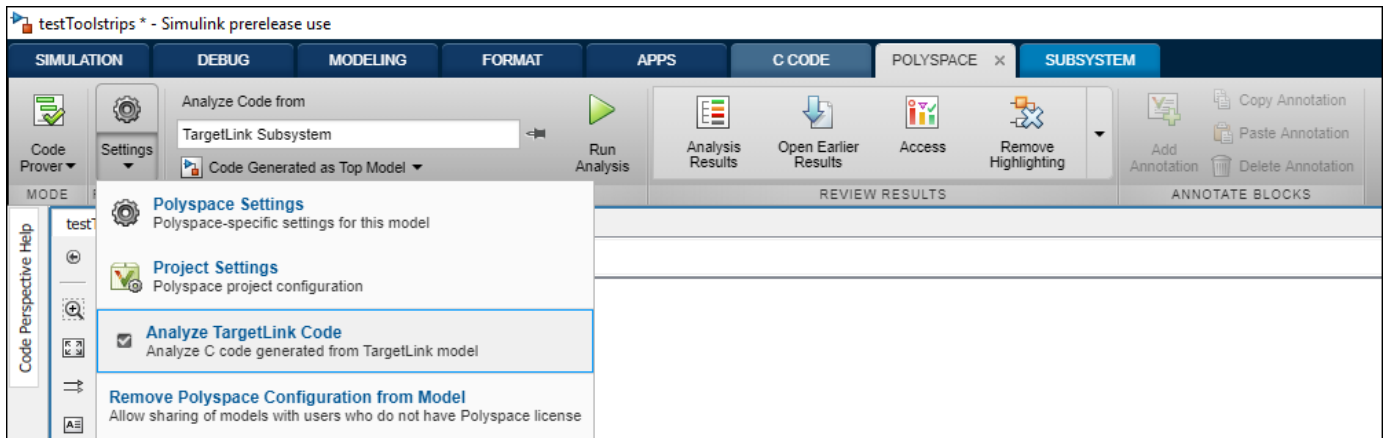
Analyze code

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 5-58
 “Default Options” on page 5-58
 “Lookup Tables” on page 5-58
 “Data Range Specification” on page 5-59
 “Code Generation Options” on page 5-59

TargetLink Support

The Windows version of Polyspace Bug Finder is supported for versions 3.5 and 4.0 of the 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” on page 10-2. 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

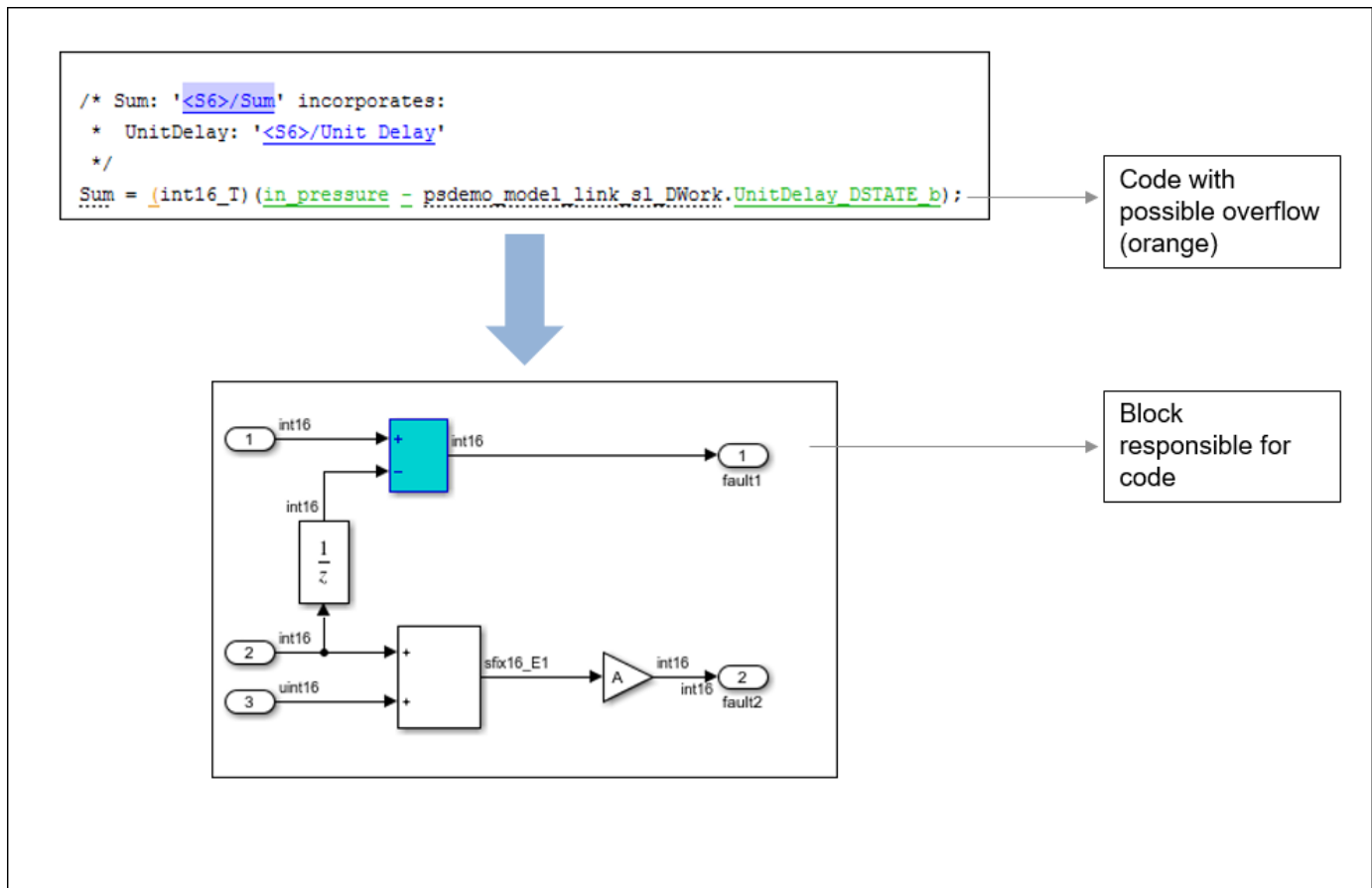
- “Run Polyspace Analysis on Code Generated with TargetLink” on page 5-56

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. 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
MATLAB/Simulink Release R2018a	Complete Integration on page 5-63	“Cross-Release Integration” on page 5-63	“Cross-Release Integration” on page 5-63	“Cross-Release Integration” on page 5-63	“Cross-Release Integration” on page 5-63	* on page 5-63	* on page 5-63
MATLAB/Simulink Release R2018b	* on page 5-63	Complete Integration on page 5-63	“Cross-Release Integration” on page 5-63	“Cross-Release Integration” on page 5-63	“Cross-Release Integration” on page 5-63	“Cross-Release Integration” on page 5-63	* on page 5-63
MATLAB/Simulink Release R2019a	* on page 5-63	* on page 5-63	Complete Integration on page 5-63	“Cross-Release Integration” on page 5-63	“Cross-Release Integration” on page 5-63	“Cross-Release Integration” on page 5-63	* on page 5-63
MATLAB/Simulink Release R2019b	* on page 5-63	* on page 5-63	* on page 5-63	Complete Integration on page 5-63	* on page 5-63	* on page 5-63	* on page 5-63
MATLAB/Simulink Release R2020a	* on page 5-63	* on page 5-63	* on page 5-63	* on page 5-63	Complete Integration on page 5-63	* on page 5-63	* on page 5-63
MATLAB/Simulink Release R2020b	* on page 5-63	* on page 5-63	* on page 5-63	* on page 5-63	* on page 5-63	Complete Integration on page 5-63	“Cross-Release Integration” on page 5-63
MATLAB/Simulink Release R2021a	* on page 5-63	* on page 5-63	* on page 5-63	* on page 5-63	* on page 5-63	* on page 5-63	Complete Integration on page 5-63

Note The empty cells (*) in the preceding table represent MATLAB and Simulink support without integration. See “Navigate Back to Model” on page 5-63.

Complete Integration

If MATLAB and Polyspace are from the same release, you can integrate them after installation by calling `polyspaceSetup`. See “Integrate Polyspace with MATLAB or Simulink from Same Release” on page 3-2.

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 “Polyspace 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 “Polyspace Analysis in MATLAB Coder”.

- Run a Polyspace analysis on handwritten C/C++ code by using MATLAB scripts.

See “Polyspace Analysis with MATLAB Scripts”.

Cross-Release Integration

You can integrate Polyspace with MATLAB or Simulink from an earlier release. See “Integrate Polyspace with MATLAB or Simulink Installation from Earlier Release” on page 3-3.

This cross-release integration offers limited functionalities. In a cross-release workflow, you can:

- Call these functions in the MATLAB Command Window to run a Polyspace analysis on C/C++ code generated by using Embedded Coder.
 - `pslinkrun`
 - `pslinkfun`
 - `pslinkoptions`

If you have MATLAB R2020b or later, use `pslinkrunCrossRelease` instead of `pslinkrun`. See “Run Polyspace on Code Generated by Using Previous Releases of Simulink” on page 5-12.

- Navigate back to Simulink model from 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 with your MATLAB/Simulink. Generally, 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.

In addition, some specific releases of MATLAB/Simulink and Polyspace do not integrate. See the table in this page.

To navigate back to model from the user interface without integrating Polyspace and MATLAB/Simulink:

- Identify the comments in your code that acts 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 3-2
- “Run Polyspace on Code Generated by Using Previous Releases of Simulink” on page 5-12

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\R2021a`. 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 3-2.

Run Polyspace Analysis

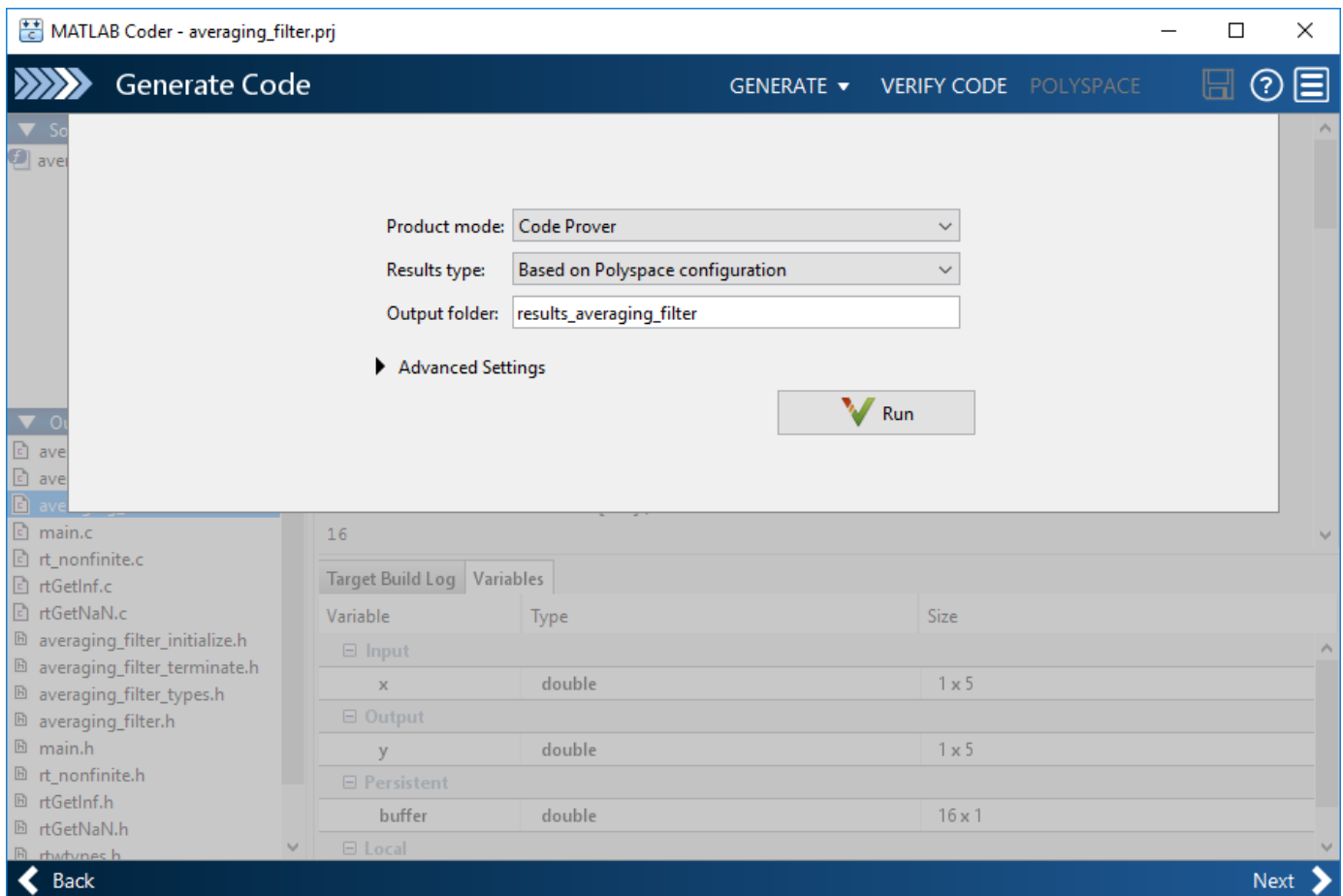
In the MATLAB Coder app, generate code from the file `averaging_filter.m` and analyze the generated code.

- 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

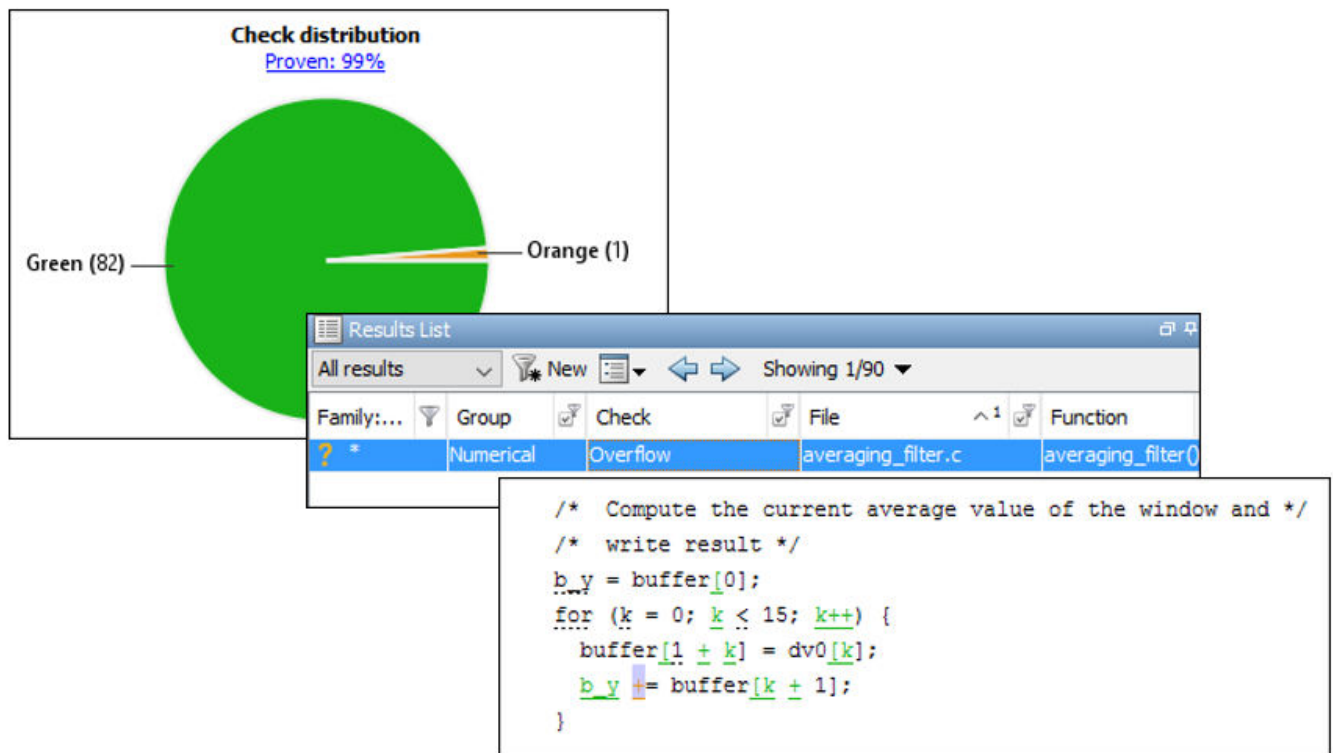
After analysis, the **Results List** pane shows a list of run-time checks. For an explanation of the result colors, see “Code Prover Result and Source Code Colors” (Polyspace Code Prover).

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**.



- 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).

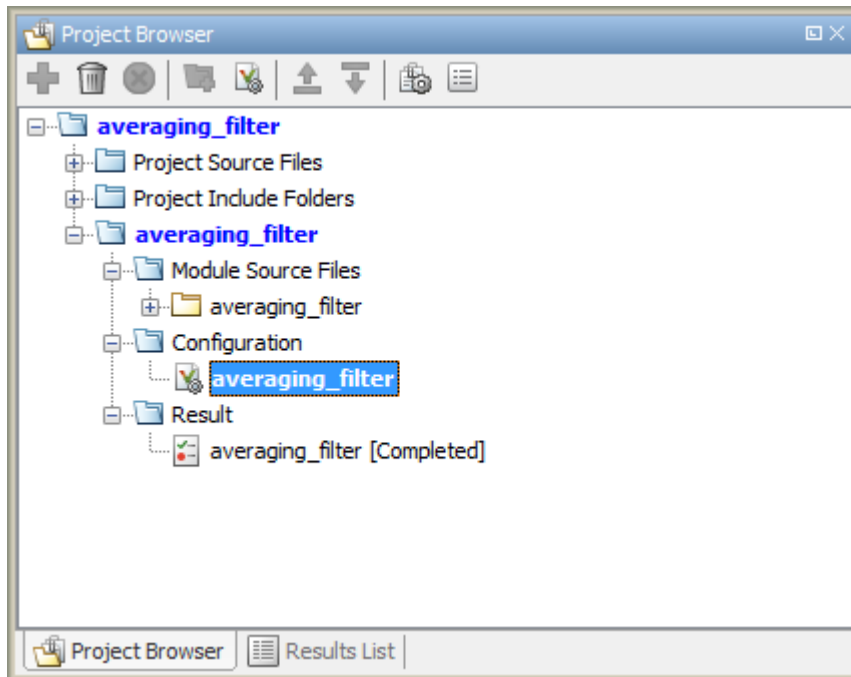
Run Analysis for Specific Design Range

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.

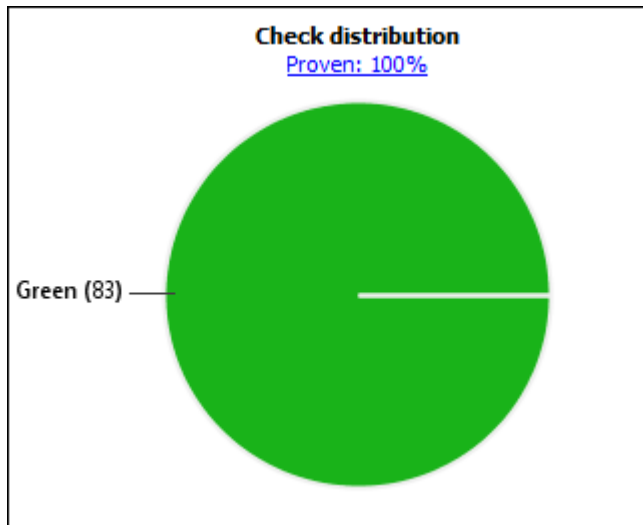
In the **Configuration** pane, on the **Inputs & Stubbing** node, set up your constraints. Click **Edit** beside **Constraint setup**. Constrain the range of the first input to [-100..100].

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 10-7.

- 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

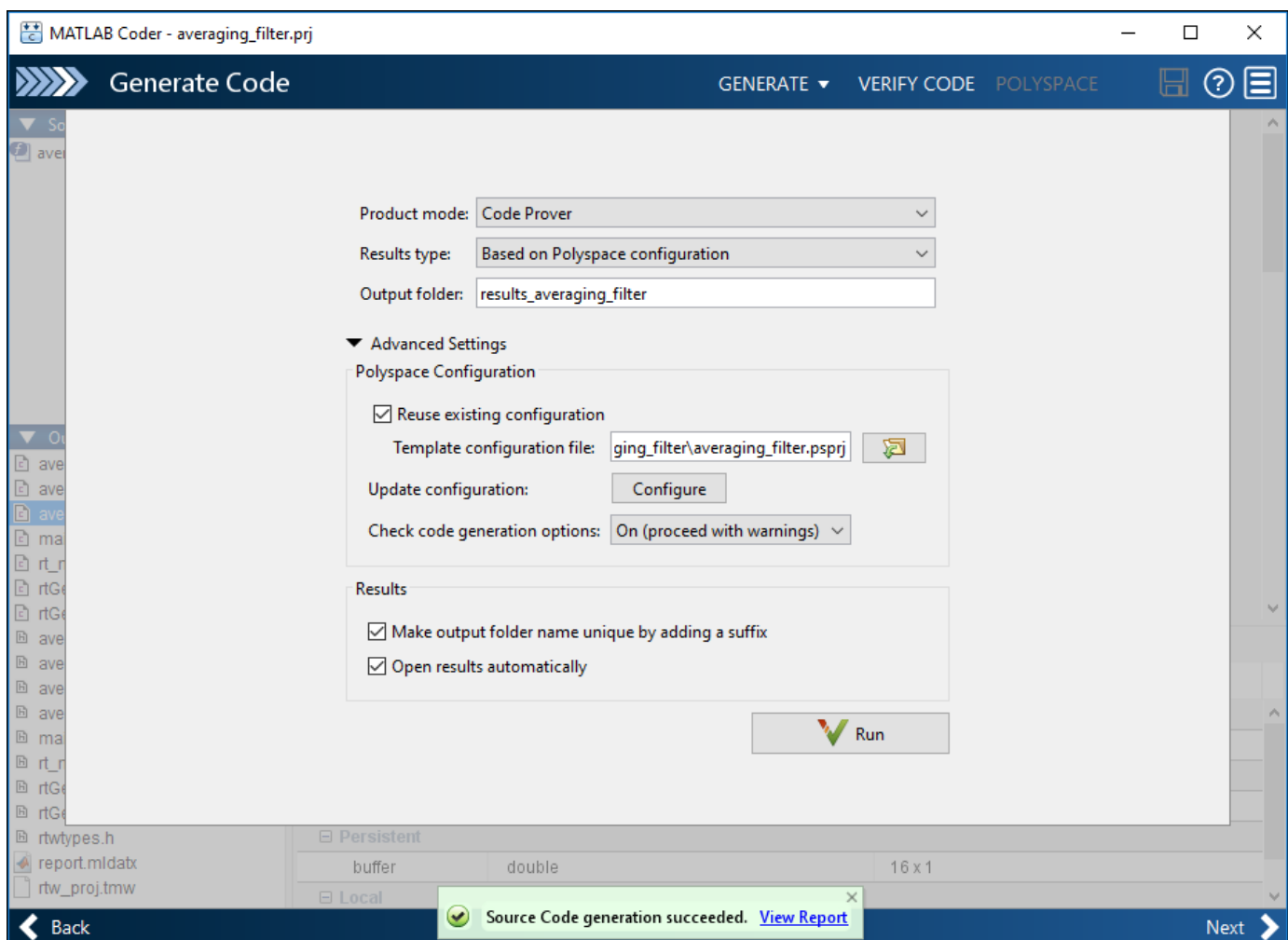
- "Configure Advanced Polyspace Options in MATLAB Coder App" on page 6-8

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 6-2.

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 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 6-2

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” (Polyspace Bug Finder Access).

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.

The screenshot shows the Eclipse IDE interface with the Polyspace plugin. The main editor displays the source code for 'My_project.c'. The code includes headers for `<stdlib.h>` and `<stdio.h>`, and defines a function `increment_content_of_address`. A defect is highlighted at line 12: `j = *pi + shift;`, with a comment: `/* Defect: Reading a freed pointer */`. The bottom panels show the 'Result Details' and 'Problems' views. The 'Result Details' view shows the defect description: 'Use of previously freed pointer (Impact: High)'. The 'Problems' view shows a table of results.

Check	Information	File	Class	File
Use of previously freed pointer	Impact: High	My_project.c	Global Scope	inc
Missing reset of freed pointer	Impact: Low	My_project.c	Global Scope	inc

After you install the Polyspace plugin, you see a **Polyspace** menu and right-click options in the **Project Explorer** to run a Polyspace analysis.

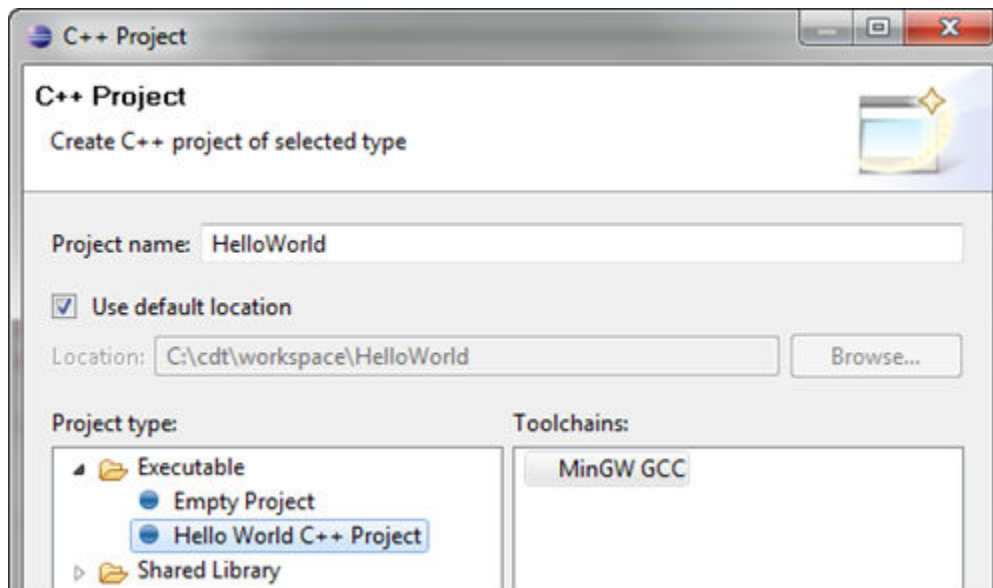
The analysis progress bar, quick run buttons and analysis results appear on specific panes. To avoid cluttering your window, you can confine these panes to the Polyspace perspective. Select **Window > Open Perspective > Other**. In the Open Perspective dialog box, select **Polyspace**. You can switch 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.

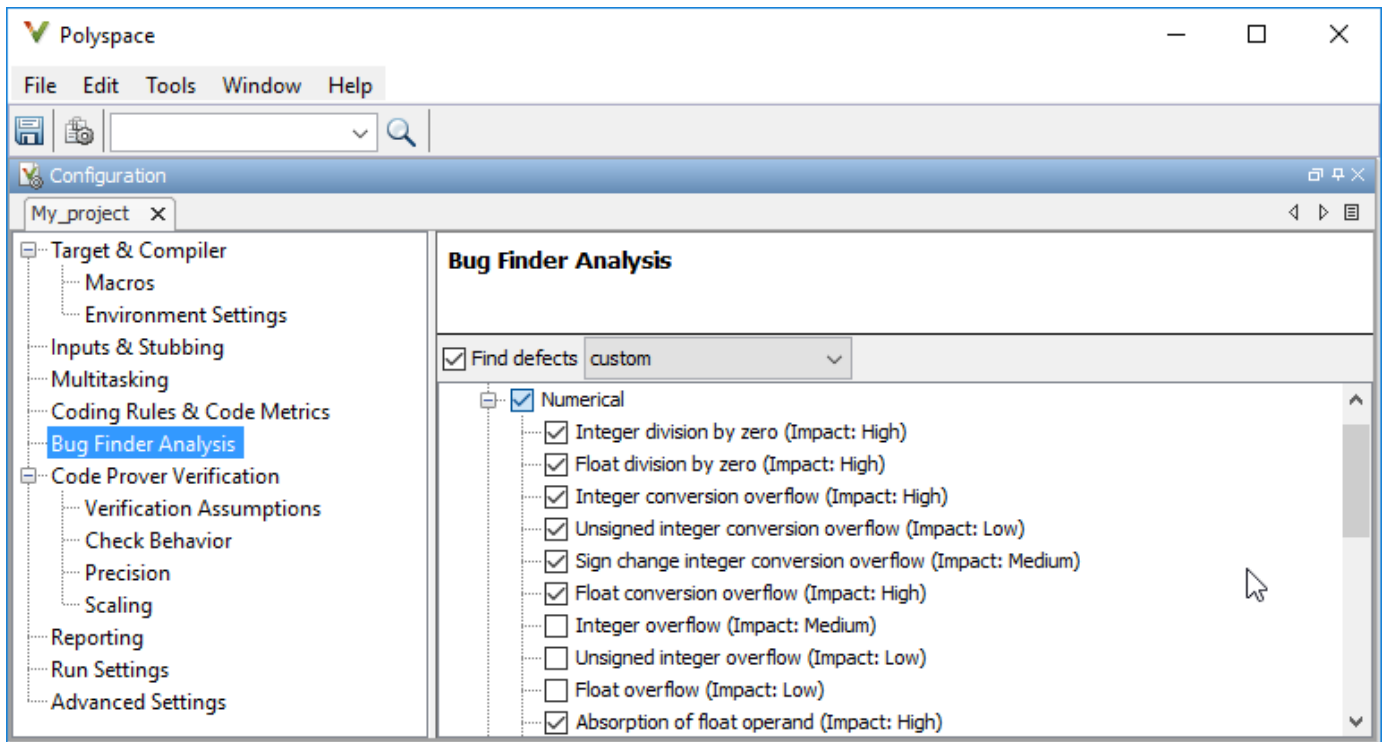


If your Eclipse project uses a build command (makefile) that has the compiler information, you must perform some additional steps to extract this information for the Polyspace analysis.

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 7-7.

- 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 “Polyspace 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.

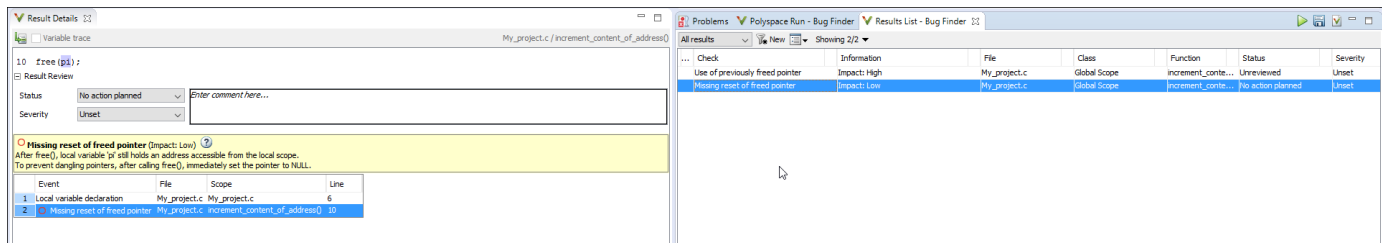
Run analysis when saving code

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.

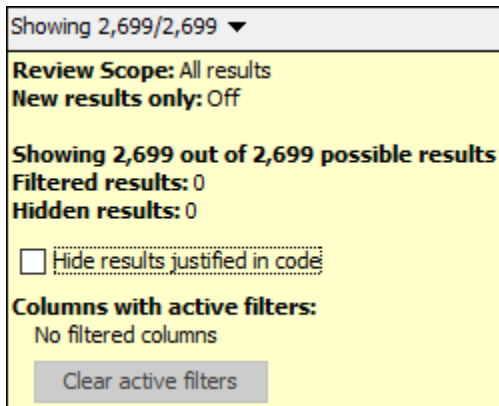
Address results

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 17-6.

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 17-6. 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 7-7
- “Interpret Bug Finder Results in Polyspace Desktop User Interface” on page 16-2
- “Address Polyspace Results Through Bug Fixes or Justifications” on page 17-2
- “Filter and Group Results in Polyspace Desktop User Interface” on page 18-2

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” (Polyspace Bug Finder Access).

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 extracts the compiler options from the project.

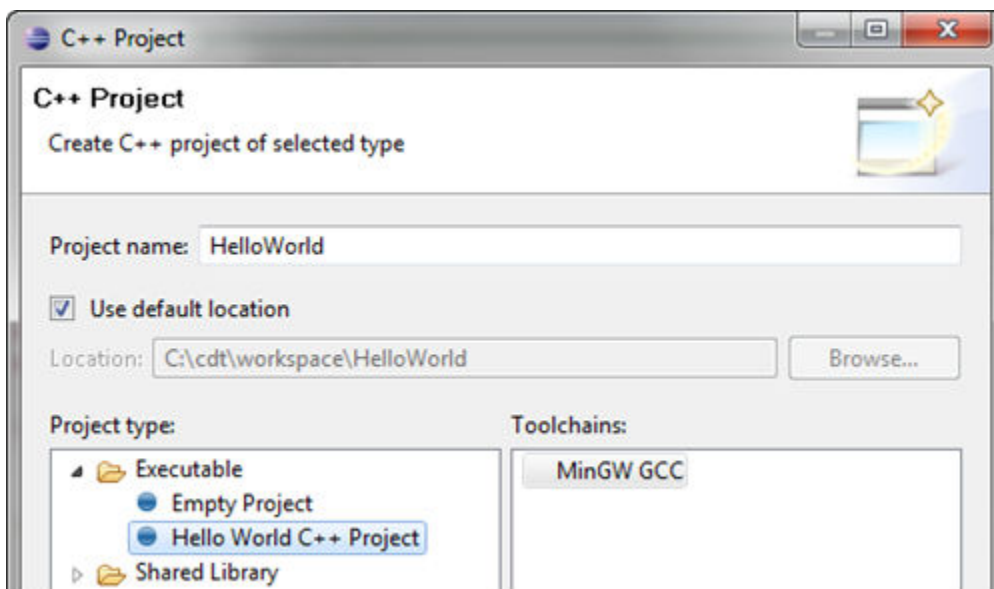
See “Eclipse Refers Directly to Your Compilation Toolchain” on page 7-7.
 - If the project refers to your compilation toolchain through a build command, the analysis cannot extract the compiler options. Trace the build command to extract the options.

See “Eclipse Uses Your Compilation Toolchain Through Build Command” on page 7-8.
- 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 7-2.

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.

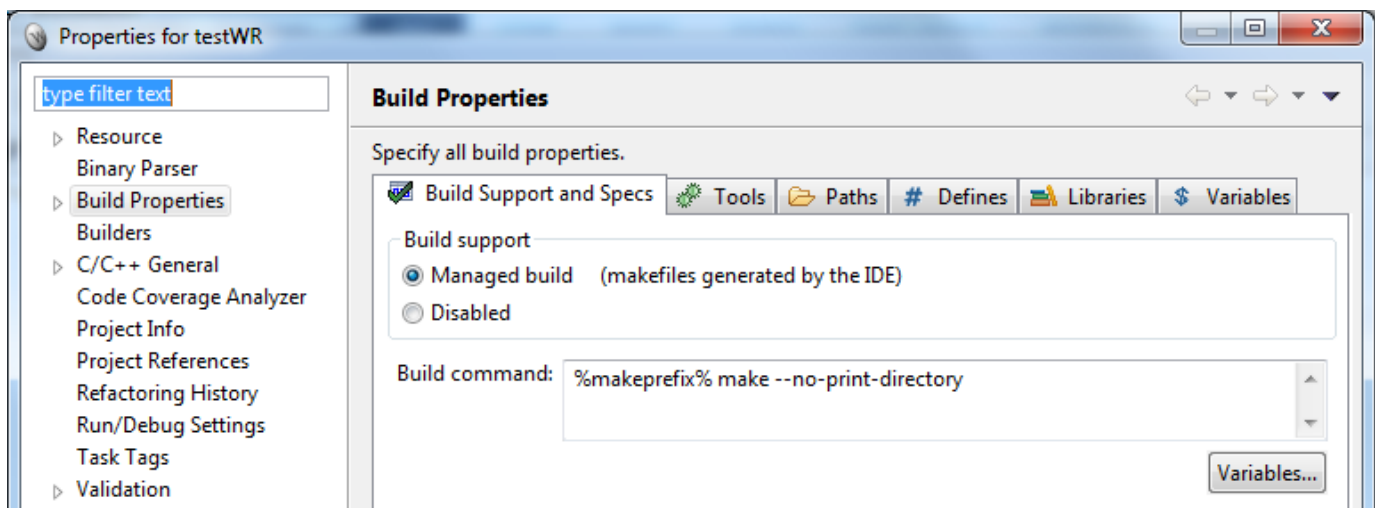
During analysis, the paths are implicitly used with the analysis option `-I`.

- 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 7-2

Configure Polyspace Analysis

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).

- Change the default behavior of run-time checkers in Code Prover.

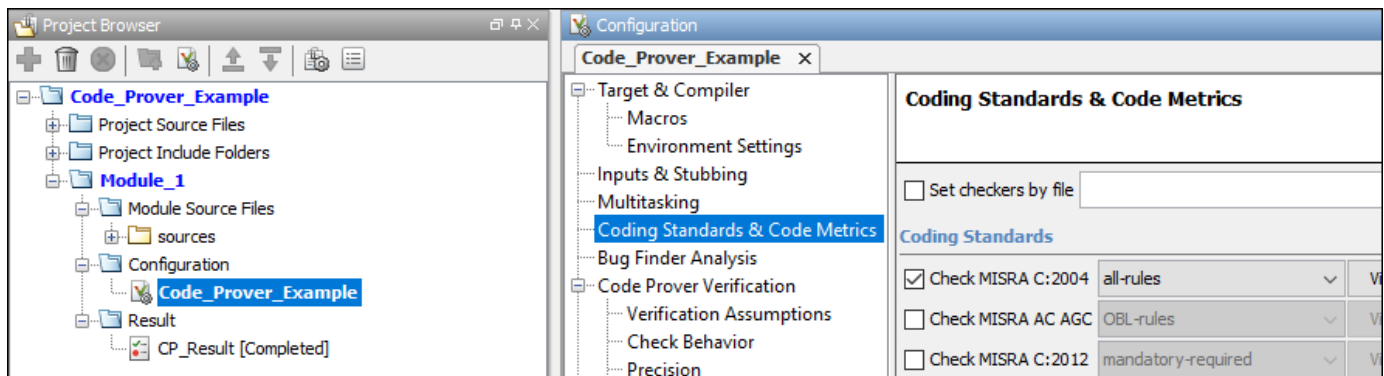
See, for instance, Overflow mode for unsigned integer (-unsigned-integer-overflows).

For the full list of analysis options, see “Analysis Options in Polyspace Bug Finder”.

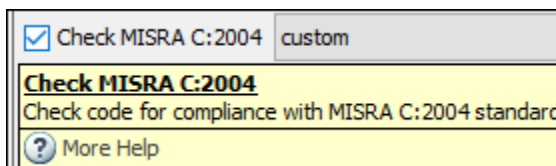
Depending on how you run Polyspace, you can configure the analysis options accordingly.

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 Polyspace Analysis on Desktop” on page 1-7.

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 2-2.

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 3-9.

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 7-2.

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 5-2
- “Configure Advanced Polyspace Options in Simulink” on page 5-45

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 6-2
- “Configure Advanced Polyspace Options in MATLAB Coder App” on page 6-8

Configure Target and Compiler Options

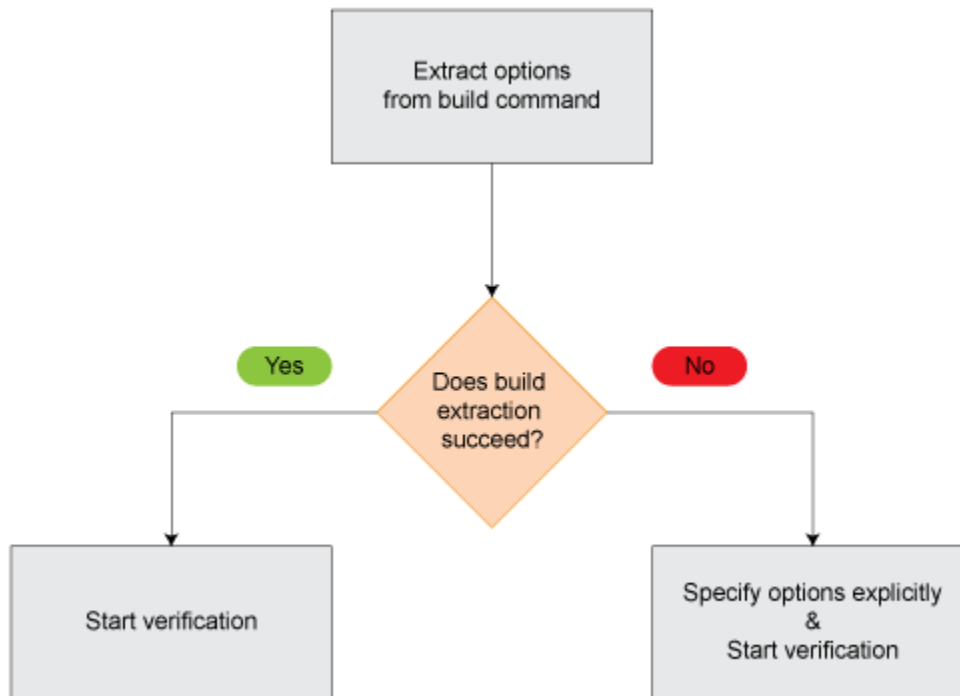
Specify Target Environment and Compiler Behavior

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. 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 User Interface" on page 1-2.
- 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” (Polyspace Bug Finder Server).

For Polyspace project creation, your build automation script (makefile) must meet certain requirements. See “Requirements for Project Creation from Build Systems” on page 9-20.

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 9-19.

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 9-19.

See Also

`C standard version (-c-version)` | `C++ standard version (-cpp-version)` | `Compiler (-compiler)` | `Preprocessor definitions (-D)` | `Source code language (-lang)` | `Target processor type (-target)`

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 9-5
- “Provide Standard Library Headers for Polyspace Analysis” on page 9-19

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

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)`.

Compiler	C Standard	C++ Standard
generic	C99	C++03
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	C11	C++14
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.

Compiler	C Standard	C++ Standard
visual9.0, visual10.0, visual11.0, visual12.0	C99	C++03
visual14.0	C99	C++14
visual15.x	C99	C++14
visual16.x	C99	C++14
keil	C99	C++03
iar	C99	C++03
armcc	C99	C++03
armclang	C11	C++03
codewarrior	C99	C++03
cosmic	C99	Not supported
diab	C99	C++03
greenhills	C99	C++03
iar-ew	C99	C++03
microchip	C99	Not supported
renesas	C99	C++03
tasking	C99	C++03
ti	C99	C++03

See Also

C standard version (-c-version) | C++ standard version (-cpp-version) | Compiler (-compiler)

More About

- “C11 Language Elements Supported in Polyspace” on page 9-7
- “C++11 Language Elements Supported in Polyspace” on page 9-9
- “C++14 Language Elements Supported in Polyspace” on page 9-12
- “C++17 Language Elements Supported in Polyspace” on page 9-15

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 9-19.
<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>CMPLEX</code> , <code>CMPLEXF</code> and <code>CMPLEXL</code> macros	Yes

See Also

C standard version (`-c-version`)

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 9-5

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 9-5
- “C++14 Language Elements Supported in Polyspace” on page 9-12
- “C++17 Language Elements Supported in Polyspace” on page 9-15

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 9-5
- “C++11 Language Elements Supported in Polyspace” on page 9-9
- “C++17 Language Elements Supported in Polyspace” on page 9-15

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: <pre>template <template <typename> typename Tmpl> struct X;</pre>	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"> <pre>namespace base::derived{ //.. }</pre> <pre>namespace { namespace derived{ //... } }</pre> 	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 9-5
- “C++11 Language Elements Supported in Polyspace” on page 9-9
- “C++14 Language Elements Supported in Polyspace” on page 9-12

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.

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 User Interface” on page 1-2.
- At the command line, use the flag `-I` with the `polyspace-bug-finder`, `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server` command..

For more information, see `-I`.

See Also

More About

- “Errors from Conflicts with Polyspace Header Files” on page 21-37

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
- Clang
- Wind River® Diab
- GNU C/C++
- IAR Embedded Workbench
- Green Hills®
- NXP CodeWarrior®
- Renesas®
- Altium® Tasking
- Texas Instruments™
- tcc - Tiny C Compiler
- Microsoft® Visual C++®

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 “Compiler Not Supported for Project Creation from Build Systems” on page 21-9.
- Contact MathWorks Technical Support. For more information, see “Contact Technical Support About Issues with Running Polyspace” on page 21-6.
- If you build your code in Cygwin™, you must be using version 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 21-16.

- 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, Polyspace cannot trace your build command. Before tracing your build command, disable the SIP feature. You can reenabte 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 User Interface” on page 1-2

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 “Errors Related to Keil or IAR Compiler” on page 21-30.

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/\s@[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 9-5. 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 9.1. 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

This example shows how to specify constraints (also known as data range specifications or DRS) on variables in your code. 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 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.

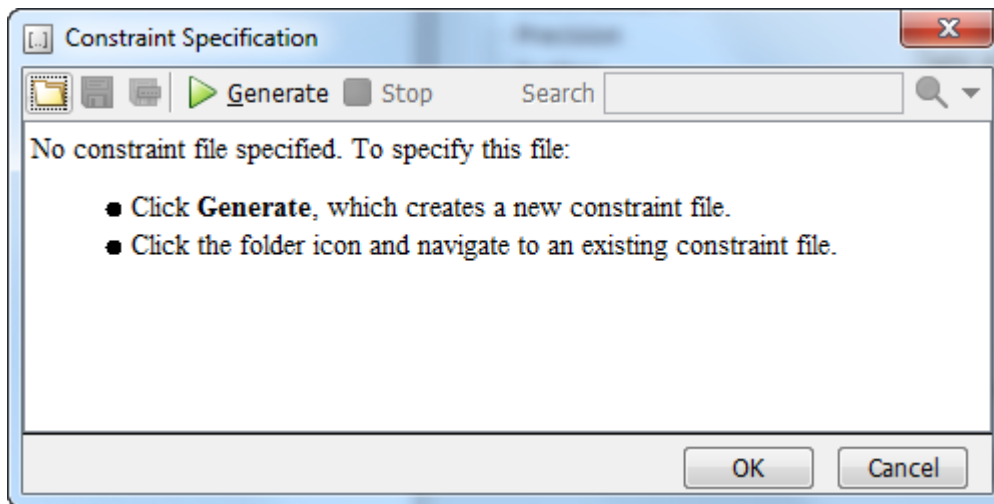
To reduce the number of such false positives, you can specify additional constraints on global variables, function inputs, and return values and modifiable arguments of stubbed functions. You save the constraints as an XML file to use them for subsequent analyses. If your source code changes, you can update the previous constraints. You do not have to create a new constraint template.

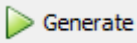
Note In Bug Finder, you can only constrain global variables. You cannot constrain function inputs or return values of stubbed functions.

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 10-7.
- 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” (Polyspace Code Prover)
- “Constrain Function Inputs” (Polyspace Code Prover)

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 **Layout > Show/Hide View > 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 10-3.
- 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 `assert` macro. For example, to constrain a variable `var` in the range `[0,10]`, you can use `assert(var >= 0 && var <=10);`.

Polyspace checks your `assert` statements to see if the condition can be false. Following the `assert` statement, Polyspace considers that the `assert` condition is true. Using `assert` statements, you can constrain your variables for the remaining code in the same scope. For examples, see [Assertion](#).

See Also

Constraint setup (-data-range-specifications)

Related Examples

- “External Constraints for Polyspace Analysis” on page 10-7
- “Constrain Global Variable Range” (Polyspace Code Prover)
- “Constrain Function Inputs” (Polyspace Code Prover)
- “XML File Format for Constraints” on page 10-19

External Constraints for Polyspace Analysis

For a more precise analysis with Polyspace, you can specify external constraints on:

- Global Variables.
- User-defined Functions.

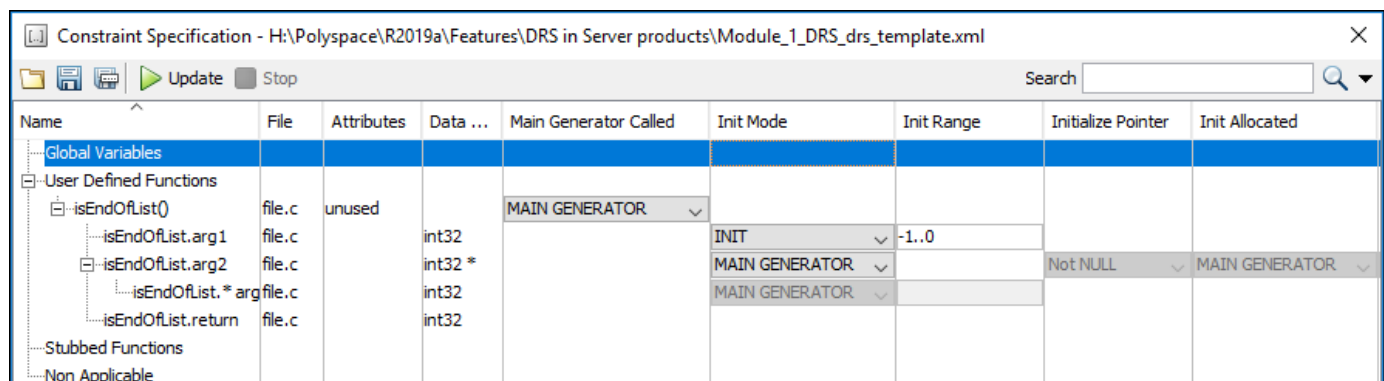
Constraints on user-defined functions do not apply to a Bug Finder analysis.

- Stubbed Functions.

Constraints on stubbed functions do not apply to a Bug Finder analysis.

For more information, see “Specify External Constraints” on page 10-2. For a partial list of limitations, see “Constraint Specification Limitations” on page 10-11.

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.



The screenshot shows the 'Constraint Specification' window for a project. The window title is 'Constraint Specification - H:\Polyspace\R2019a\Features\DRS in Server products\Module_1_DRS_drs_template.xml'. It has a search bar and 'Update' and 'Stop' buttons. The main content is a table with the following columns: Name, File, Attributes, Data ..., Main Generator Called, Init Mode, Init Range, Initialize Pointer, and Init Allocated. The table is organized into sections: Global Variables, User Defined Functions, Stubbed Functions, and Non Applicable. Under 'User Defined Functions', there is a tree view for 'isEndOfList()' with sub-entries 'isEndOfList.arg1', 'isEndOfList.arg2', 'isEndOfList.* arg', and 'isEndOfList.return'. The table rows correspond to these entries, with some cells containing dropdown menus for 'Main Generator Called', 'Init Mode', 'Initialize Pointer', and 'Init Allocated'. The row for 'isEndOfList.* arg' is highlighted in bold in the original image.

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 explains 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> <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</code> (C) or <code>-main-generator-calls</code> (C++) 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>

Column	Settings
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 <code>NULL</code>, 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>
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 enum variables, you cannot specify ranges directly using the enumerator constants. Instead use the values represented by the constants.</p> <p>For enum 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 enum 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>

Column	Settings
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. <hr/> <p>Note Not applicable for C++ projects. See “Constraint Specification Limitations” on page 10-11.</p> <hr/> <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. <hr/> <p>Note Not applicable for C++ projects. See “Constraint Specification Limitations” on page 10-11.</p> <hr/> <p>XML File Entry:</p> <pre><pointer name="arg1" init_pointed="MAIN_GENERATOR" ...></pre>

Column	Settings
# Allocated Objects	<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>void func(int *ptr) { assert(ptr[0]==1); assert(ptr[1]==1); }</pre> <p>If you specify these constraints:</p> <ul 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 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> <hr/> <p>Note Not applicable for C++ projects. See “Constraint Specification Limitations” on page 10-11.</p> <hr/> <p>XML File Entry:</p> <pre><pointer name="arg1" number_allocated="10" ...></pre>
Global Assert	<p>Specifies whether to perform an assert check on the variable at global initialization, and after each assignment.</p> <hr/> <p>XML File Entry:</p> <pre><scalar name="glob" global_assert="YES" ...></pre>
Global Assert Range	<p>Specifies the minimum and maximum values for the range you want to check.</p> <hr/> <p>XML File Entry:</p> <pre><scalar name="glob" assert_range="0..200" ...></pre>
Comment	<p>Remarks that you enter, for example, justification for your DRS values.</p> <hr/> <p>XML File Entry:</p> <pre><scalar name="glob" comment="Speed Range" ...></pre>

Constraint Specification Limitations

You cannot specify these constraints:

- *C++ Pointers cannot be constrained:*

In C++, you cannot constrain pointer arguments of functions. Functions that have pointer arguments only do not appear in the constraint specification interface.

Because of polymorphism, a C++ pointer 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.

- *Constraints cannot be relations:*

You cannot specify a constraint that relates the return value of a function to its inputs. You can only specify a constant range for the constraints.

- *Multiple ranges not possible:*

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].

See Also

More About


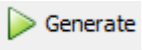
- “Specify External Constraints” on page 10-2

Constrain Global Variable Range

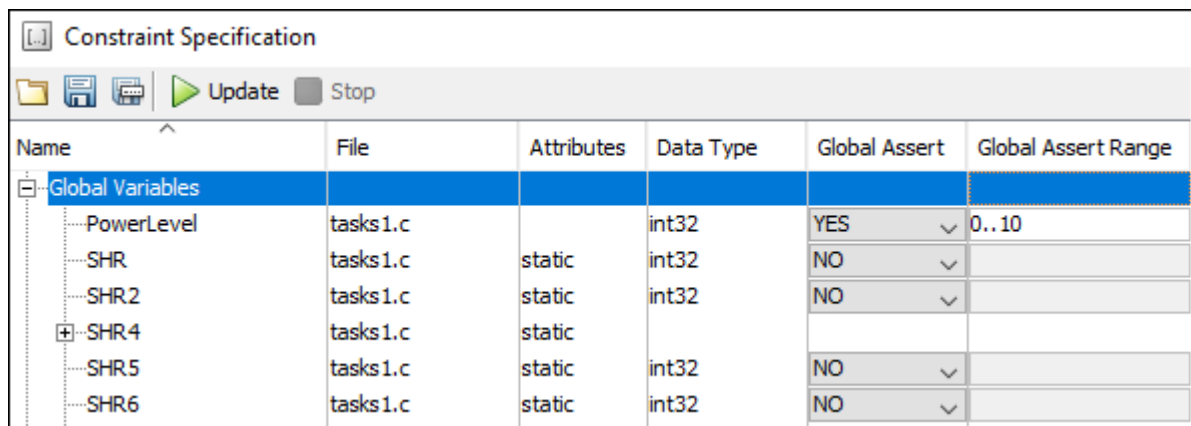
You can impose constraints (also known as data range specifications or DRS) on the range of a global variable and check with Code Prover whether write operations on the variable violate the constraint. For the general workflow, see “Specify External Constraints” on page 10-2.

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” on page 10-2. 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” on page 10-2
- “External Constraints for Polyspace Analysis” on page 10-7
- “Constrain Function Inputs” (Polyspace Code Prover)

Constrain Function Inputs

For a more precise Code Prover analysis, you can specify constraints (also known as data range specifications or DRS) on function inputs. Code Prover checks your function definition for run-time errors with respect to the constrained inputs. For the general workflow, see “Specify External Constraints” on page 10-2.

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) {
    .
    .
}
```

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.

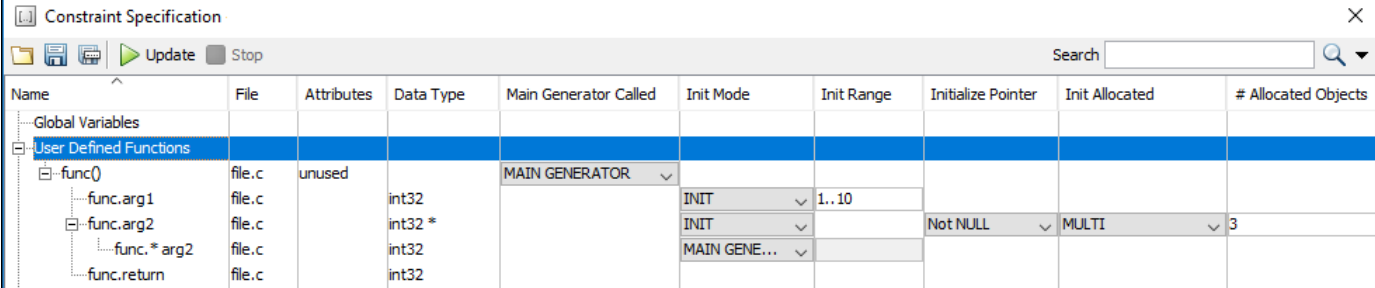
- 3 Expand the node for each function.

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 10-7.

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**.
- 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 GENERATOR				
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” on page 10-2. 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” on page 10-2
- “External Constraints for Polyspace Analysis” on page 10-7
- “Constrain Global Variable Range” (Polyspace Code Prover)

XML File Format for 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” on page 10-2.

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” (Polyspace Code Prover)
- “Constrain Function Inputs” (Polyspace Code Prover)

For another explanation of what the XML tags mean, see “External Constraints for Polyspace Analysis” on page 10-7.

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 10-23.

- **(*****)** — 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 PERMANENT	YES NO			Main generator dependent
		Volatile scalar	PERMANENT	disabled			PERMANENT min..max
		Extern scalar	INIT PERMANENT	YES NO			INIT min..max
	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

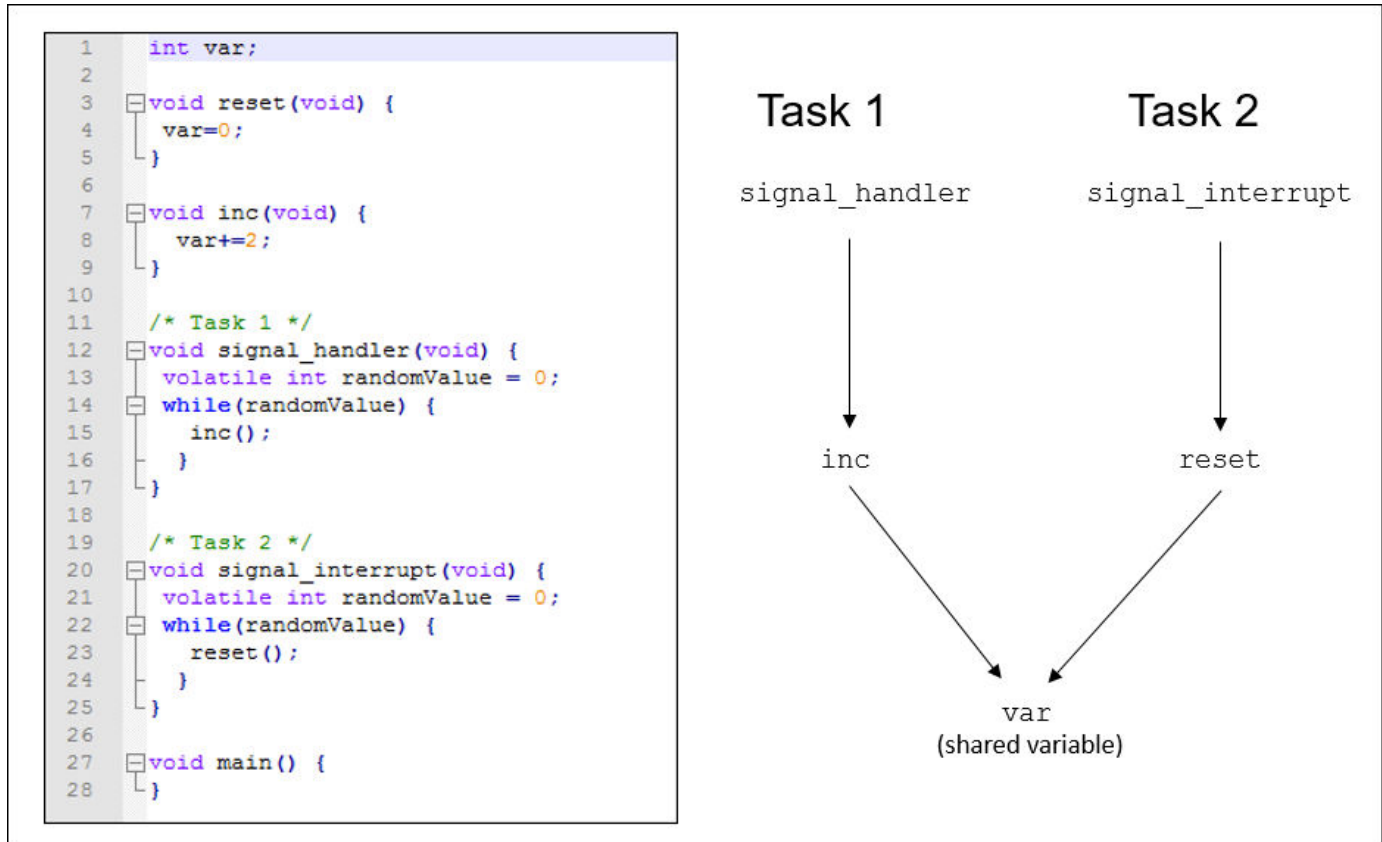
More About

- “Specify External Constraints” on page 10-2
- “Constrain Global Variable Range” (Polyspace Code Prover)
- “Constrain Function Inputs” (Polyspace Code Prover)

Configure Multitasking Analysis

Analyze Multitasking Programs in Polyspace

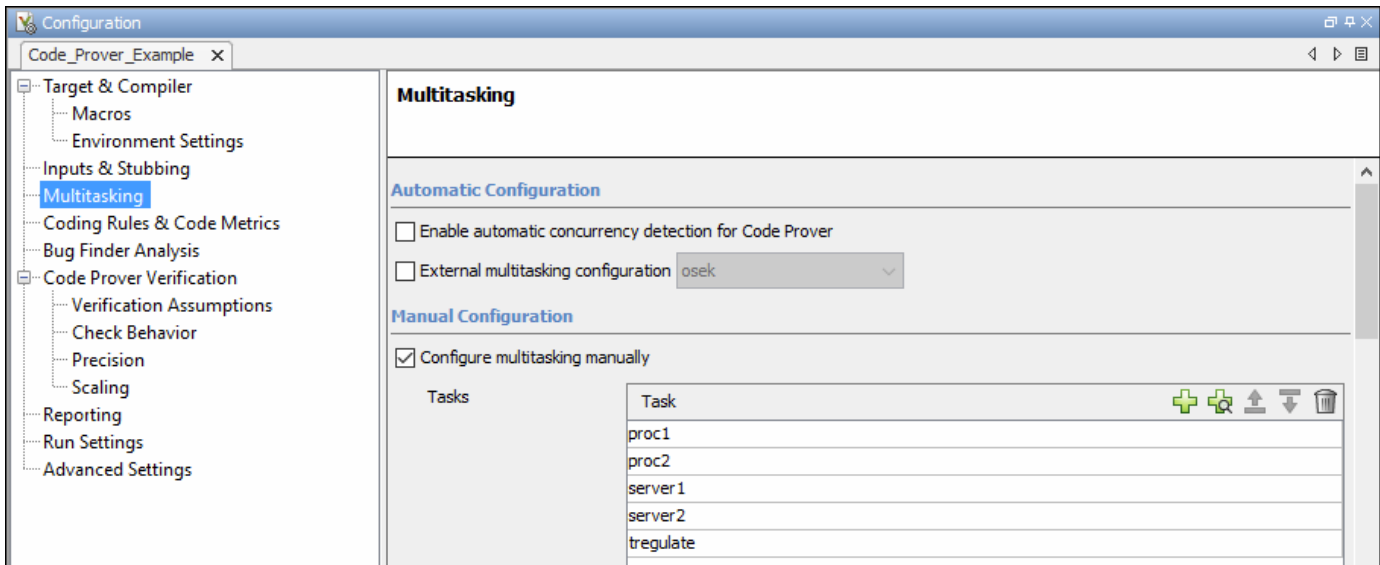
With Polyspace, you can analyze programs where multiple threads (tasks) run concurrently.



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:

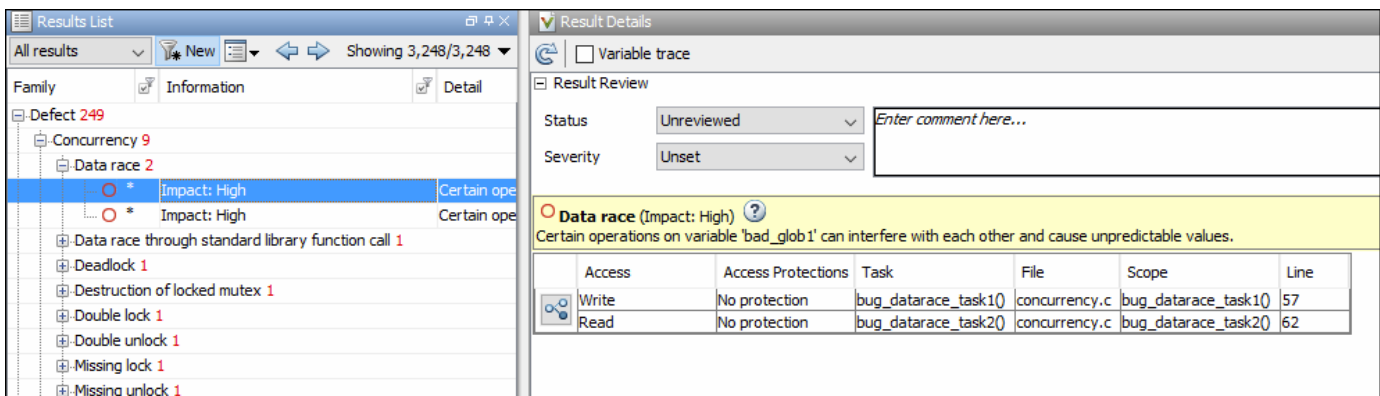
- In Bug Finder, the analysis detects them and extracts your multitasking model from the code.
- In Code Prover, you must enable this automatic detection explicitly.

See “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 11-5.

Alternatively, define your multitasking model through the analysis options. In the user interface, the options are on the **Multitasking** node in the **Configuration** pane. For more information, see “Configuring Polyspace Multitasking Analysis Manually” on page 11-16.

Review Analysis Results

Bug Finder




The Bug Finder analysis shows concurrency defects such as data races and deadlocks. See “Concurrency Defects”.

Code Prover

The screenshot displays the Code Prover interface. The left pane, titled "Results List", shows a tree view of analysis results. Under "Global Variable", the "Potentially unprotected variable" category is expanded, listing several variables: "Variable: PowerLevel", "Variable: SHR4", "Variable: SHR2", "Variable: SHR", and "Variable: SHR5". The "Variable: PowerLevel" entry is selected and highlighted in blue. The right pane, titled "Result Details", shows the details for the selected variable. It includes a "Result Review" section with "Status" set to "Unreviewed" and "Severity" set to "Unset". Below this, a yellow warning message 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." A table below the message lists the events for the variable:

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. 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.

See Also

More About

- “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 11-5
- “Configuring Polyspace Multitasking Analysis Manually” on page 11-16
- “Protections for Shared Variables in Multitasking Code” on page 11-20

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 11-2.

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 11-16.

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 Using Configuration Template” on page 1-16. 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 11-11.

C11

Thread creation: `thr_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 `phil01`, `phil02`, `phil03`, `phil04`, and `phil05` 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

-code-behavior-specifications | Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)

More About

- “Analyze Multitasking Programs in Polyspace” on page 11-2
- “Configuring Polyspace Multitasking Analysis Manually” on page 11-16

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 11-5.

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 11-2.

Specify Options for Multitasking Analysis

Use these options to specify cyclic tasks, interrupts and protections for shared variables. In the Polyspace user interface, the options are on the **Multitasking** node in the **Configuration** pane.

- **Entry points (-entry-points)**: Specify noncyclic entry point functions.
Do not specify `main`. Polyspace implicitly considers `main` as an entry point function.
- **Cyclic tasks (-cyclic-tasks)**: Specify functions that are scheduled at periodic intervals.
- **Interrupts (-interrupts)**: Specify functions that can run asynchronously.
- **Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)**: Specify functions that disable and reenable interrupts (Bug Finder only).
- **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.
- **-preemptable-interrupts**: Specify functions that have lower priority than interrupts, but higher priority than tasks (preemptable or non-preemptable).
Only the Bug Finder analysis considers priorities.
- **-non-preemptable-tasks**: Specify functions that have higher priority than tasks, but lower priority than interrupts (preemptable or non-preemptable).
Only the Bug Finder analysis considers priorities.

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`.

All tasks and interrupts can interrupt each other.

The Bug Finder analysis considers priorities of tasks. A function that you specify as a task cannot interrupt a function that you specify as an interrupt because an interrupt has higher priority.

The Code Prover analysis considers that all tasks and interrupts can interrupt each other.

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();  
    if(randomValue)  
        inc();  
    reset();  
}  
}
```

See Also

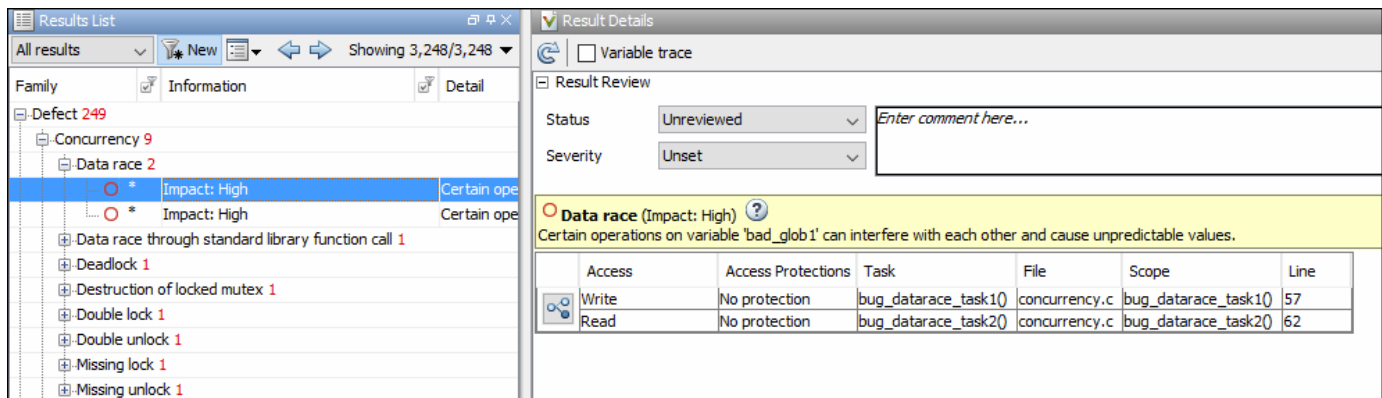
More About

- “Analyze Multitasking Programs in Polyspace” on page 11-2
- “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 11-5

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_task1()	concurrency.c	bug_datarace_task1()	57
Read	No protection	bug_datarace_task2()	concurrency.c	bug_datarace_task2()	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 11-5.

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 the checker `Data race including atomic operations` ignores the difference in priorities and continues to

show the data race. See also “Define Preemptable Interrupts and Nonpreemptable Tasks” on page 11-27.

Code Prover does not consider priorities of tasks. Therefore, Code Prover still shows `shared_var` as a potentially unprotected global variable.

Protect By Disabling Interrupts

In a Bug Finder analysis, you can 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.

See Also

More About

- “Analyze Multitasking Programs in Polyspace” on page 11-2
- “Define Atomic Operations in Multitasking Code” on page 11-24

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 11-25.

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, enable `Data race including atomic operations`.

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** (Bug Finder only)

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 Preemptable Interrupts and Nonpreemptable Tasks” on page 11-27.

- **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 11-20.

See Also

Critical section details (`-critical-section-begin -critical-section-end`) |
Cyclic tasks (`-cyclic-tasks`) | Interrupts (`-interrupts`) | Temporally exclusive
tasks (`-temporal-exclusions-file`)

More About

- “Analyze Multitasking Programs in Polyspace” on page 11-2
- “Protections for Shared Variables in Multitasking Code” on page 11-20

Define Preemptable Interrupts and Nonpreemptable Tasks

Bug Finder detects data races between concurrent tasks. Using Bug Finder analysis options, you can fix data race detection 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 `Entry points (-entry-points)` and `-non-preemptable-tasks`.

- Cyclic tasks (preemptable): Use option `Cyclic tasks (-cyclic-tasks)`.

You can also define noncyclic tasks with the option `Entry points (-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`.
- Enable the checker `Data race including atomic operations` (not enabled by default). Use the option `Find defects (-checkers)`.

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.

See Also

Polyspace Analysis Options

`-non-preemptable-tasks` | `-preemptable-interrupts` | `Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)`

Polyspace Results

`Data race` | `Data race including atomic operations`

More About

- “Analyze Multitasking Programs in Polyspace” on page 11-2
- “Protections for Shared Variables in Multitasking Code” on page 11-20
- “Define Atomic Operations in Multitasking Code” on page 11-24

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 11-24.

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 11-20

Configure Coding Rules Checking and Code Metrics Computation

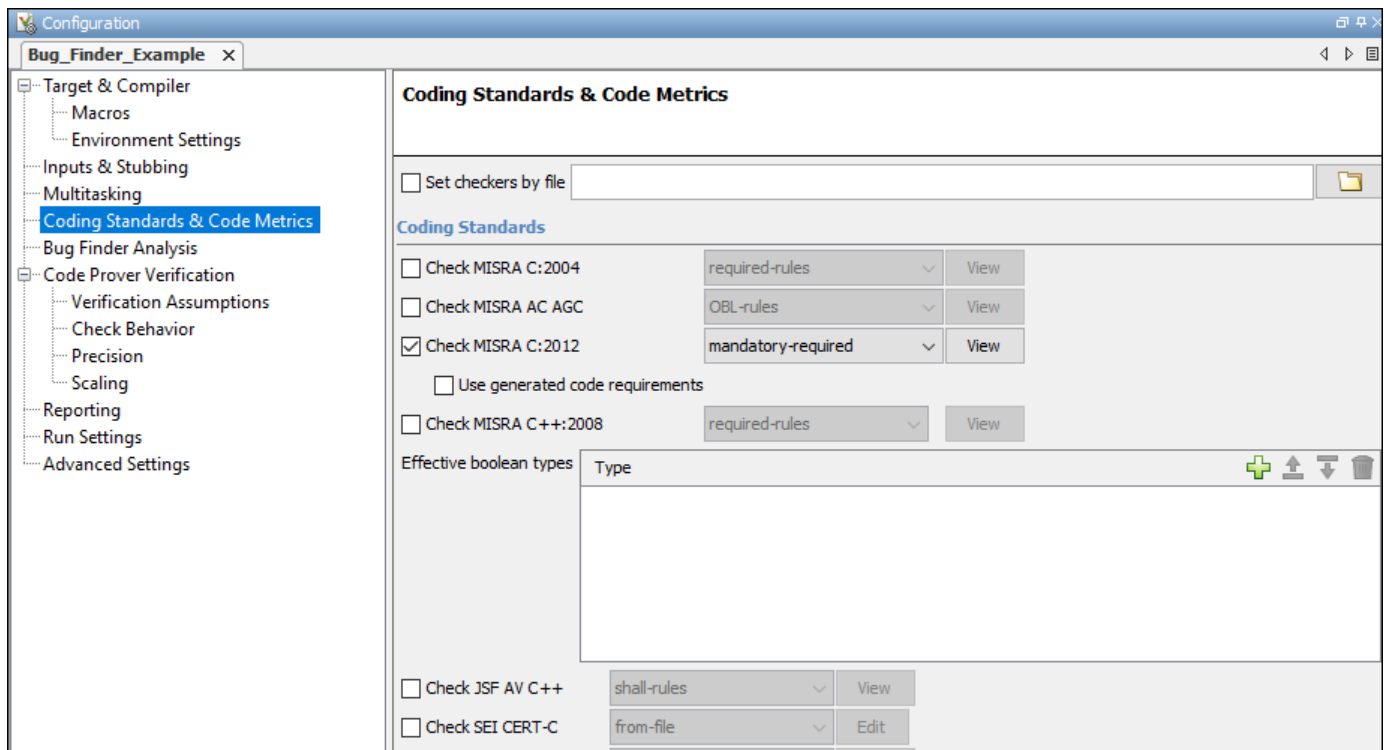
Check for Coding Standard Violations

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 the violations of these rules:

- 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

Configure Coding Rules Checking



Specify Standard and Predefined Checker Subsets

Specify the coding rules through Polyspace analysis options. When you run Bug Finder or Code Prover, 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)

For C++ code, use one of these options:

- Check MISRA C++ rules (-misra-cpp)
- Check JSF++ rules (-jsf-coding-rules)
- Check AUTOSAR C++ 14 (-autosar-cpp14)
- Check SEI CERT-C++ (-cert-cpp)
- Check Guidelines (-guidelines)

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 12-47.

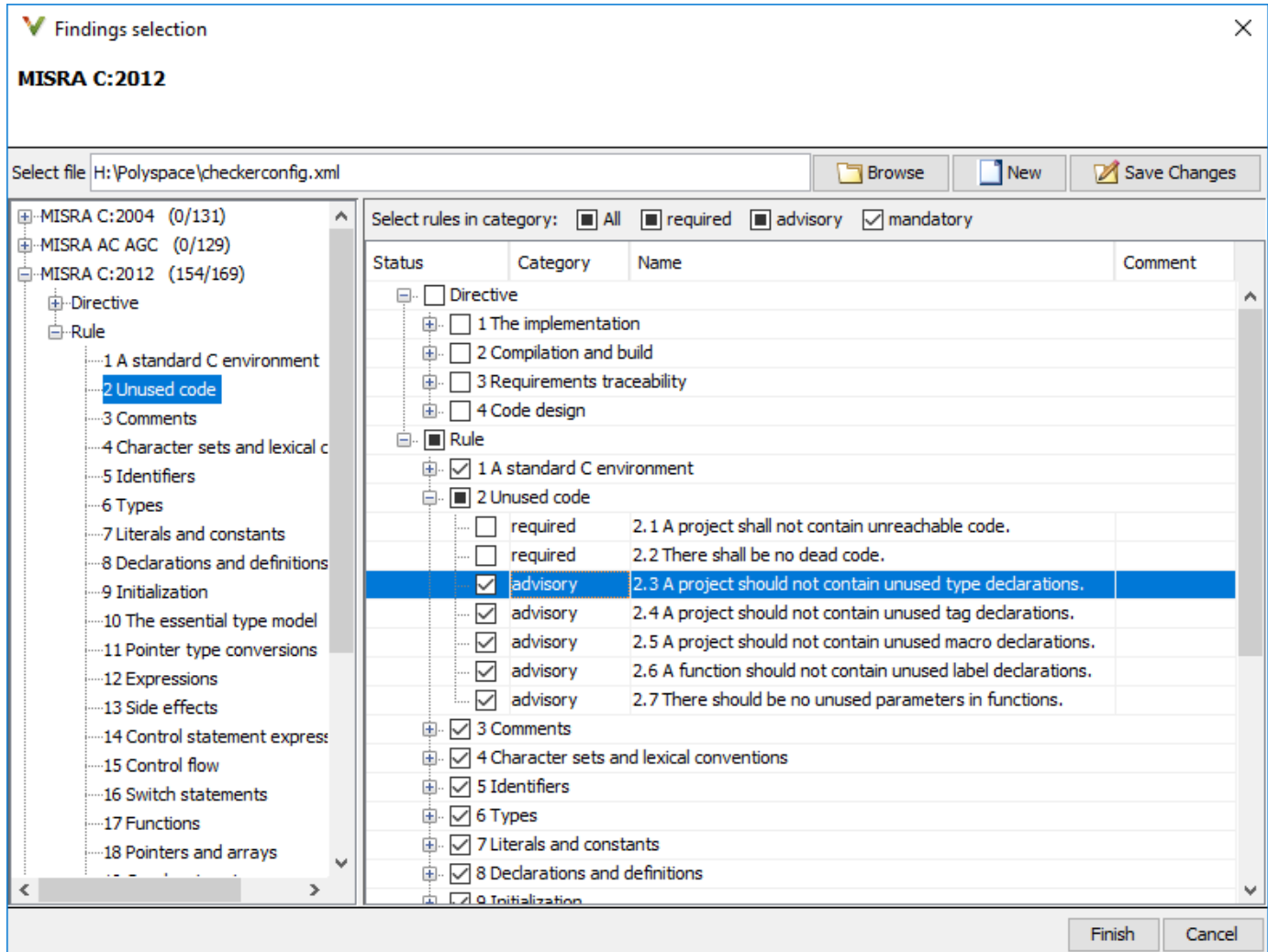
Customize Checker Subsets

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 **Findings 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 .

- 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\polyspace_bug_finder_server\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\R2019a`.

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"
- "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

To check for coding standards only:

- In Bug Finder, disable checking of defects. Use the option `Find defects (-checkers)`.
- In Code Prover, check for source compliance only. Use the option `Verification level (-to)`.

These rules are checked in the later stages of a Code Prover analysis: MISRA C:2004 rules 9.1, 13.7, and 21.1, and MISRA C:2012 rules 2.2, 9.1, 14.3, and 18.1. If you stop Code Prover at source compliance checking, the analysis might not find all violations of these rules. You can also see a difference in results based on your choice for the option `Verification level (-to)`. For example, it is possible that Code Prover suspects in the first pass that a variable may be uninitialized but proves in the second pass that the variable is initialized. In that case, you see a violation of MISRA C:2012 Rule 9.1 in the first pass but not in the second pass.

Review Coding Rule Violations

Result Details

Variable trace

Result Review

Status: To fix

Severity: Medium

MISRA C:2012 5.1 (Required) External identifiers shall be distinct.
External function `demo_corrected_sighandlerasynsafestrict` conflicts with the external identifier `demo_corrected_sighandlerasynsafe` (`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

Configuration | Result Details

Source

```

programming.c x
1225
1226 void corrected_sighandlerasynsafestrict(int signum) {
1227     int s0 = signum; /* Fix: avoid raise() */
1228 }
1229
1230 int demo_corrected_sighandlerasynsafestrict(void) {
1231     if (signal(SIGTERM, demo_term_handler) == SIG_ERR) {
1232         /* Handle error */
1233     }
1234     if (signal(SIGINT, corrected_sighandlerasynsafestrict) == 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”.

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 16-2
- “Address Polyspace Results Through Bug Fixes or Justifications” on page 17-2
- “Filter and Group Results in Polyspace Desktop User Interface” on page 18-2
- “Generate Reports” on page 19-2

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” on page 12-49.

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 12-52. 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 Coding Standard Violations” on page 12-2.
- 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” (Polyspace Bug Finder Access).
 - 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” (Polyspace Bug Finder Access).
 - 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” (Polyspace Bug Finder Access).
 - 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” (Polyspace Bug Finder Access).

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 = CalculateV(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 17-6. 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 12-16
“Rules in SQO-Subset2” on page 12-17

Rules in SQO-Subset1

In Polyspace Code Prover, the following set of coding rules will typically reduce the number of unproven results.

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 <i>static</i> 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.
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.
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 <i>for</i> statement shall not contain any objects of floating type.
13.5	The three expressions of a <i>for</i> statement shall be concerned only with loop control.
14.4	The <i>goto</i> 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.
18.3	An area of memory shall not be reused for unrelated purposes.
18.4	Unions shall not be used.

Rule number	Description
20.4	Dynamic heap memory allocation shall not be used.

Note Polyspace software does not check MISRA rule **18.3**.

Rules in SQO-Subset2

Good design practices generally lead to less code complexity, which can reduce the number of unproven results in Polyspace Code Prover. 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	<i>typedefs</i> 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 <i>static</i> 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.
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 = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized
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	Bitwise operations shall not be performed on signed integer types
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 void.
11.3	A cast should not be performed between a pointer type and an integral type.
11.5	Type casting from any type to or from pointers shall not be used
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

Rule number	Description
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 !)
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
12.12	The underlying bit representations of floating-point values shall not be used.
13.1	Assignment operators shall not be used in expressions that yield 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 <i>for</i> statement shall not contain any objects of floating type.
13.5	The three expressions of a <i>for</i> statement shall be concerned only with loop control.
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
14.4	The <i>goto</i> 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 <i>switch</i> , <i>while</i> , <i>do while</i> or <i>for</i> statement shall be a compound statement
14.10	All <i>if else if</i> constructs should contain a final <i>else</i> clause
15.3	The final clause of a <i>switch</i> statement shall be the <i>default</i> 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 parenthesized 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.

Rule number	Description
18.3	An area of memory shall not be reused for unrelated purposes.
18.4	Unions shall not be used.
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
19.9	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 # 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 ## preprocessor 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 Coding Rule Violations” on page 5-19

Software Quality Objective Subsets (AC AGC)

In this section...
“Rules in SQO-Subset1” on page 12-20
“Rules in SQO-Subset2” on page 12-20

Rules in SQO-Subset1

In Polyspace Code Prover, the following set of coding rules will typically reduce the number of unproven results.

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 <i>static</i> 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.
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.
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.
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.
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
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.

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, which can reduce the number of unproven results in Polyspace Code Prover. 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.

Rule number	Description
6.3	<i>typedefs</i> 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 <i>static</i> 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.
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
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 void.
11.3	A cast should not be performed between a pointer type and an integral type.
11.5	Type casting from any type to or from pointers shall not be used
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits
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
12.12	The underlying bit representations of floating-point values 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.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration
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 parenthesized parameter list, which may be empty
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
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.
19.9	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 # or ##

Rule number	Description
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.
20.3	The validity of values passed to library functions shall be checked.

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 Coding Rule Violations” on page 5-19

Software Quality Objective Subsets (C:2012)

In this section...
“Guidelines in SQO-Subset1” on page 12-23
“Guidelines in SQO-Subset2” on page 12-24

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 following set of MISRA C:2012 coding guidelines typically reduces the number of unproven results in Polyspace Code Prover.

Rule	Description
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 <starg.h> shall not be used
17.2	Functions shall not call themselves, either directly or indirectly
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

Rule	Description
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 <stdlib.h> shall not be used

Guidelines in SQO-Subset2

Good design practices generally lead to less code complexity, which can reduce the number of unproven results in Polyspace Code Prover. 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
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
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

Rule	Description
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 <starg.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
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 Coding Rule Violations” on page 5-19

Software Quality Objective Subsets (C++)

In this section...

“SQO Subset 1 - Direct Impact on Selectivity” on page 12-26

“SQO Subset 2 - Indirect Impact on Selectivity” on page 12-27

SQO Subset 1 - Direct Impact on Selectivity

The following set of MISRA C++ coding rules will typically improve the number of unproven results in Polyspace Code Prover.

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	The One Definition Rule shall not be violated.
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 nonvirtual 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 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).
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, which can improve the number of unproven results in Polyspace Code Prover. 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.

MISRA C++ Rule	Description
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.
3-4-1	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.

MISRA C++ Rule	Description
5-2-11	The comma operator, && operator and the operator shall not be overloaded.
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.

MISRA C++ Rule	Description
10-1-3	An accessible base class shall not be both virtual and nonvirtual 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.
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 Coding Rule Violations” on page 5-19

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	Check rules that apply only to single translation units. If you detect only coding rule violations and select this subset, a Bug Finder analysis stops after the compilation phase.
system-decidable-rules	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. If you detect only coding rule violations and select this subset, a Bug Finder analysis stops after the linking phase.

See also “Check for Coding Rule Violations” on page 5-19.

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 Coding Rule Violations” on page 5-19

Create Custom Coding Rules

This example shows how to create a custom coding rules file. You can use this file to check names or text patterns in your source code against custom rules that you specify. For each rule, you specify a pattern in the form of a regular expression. The software compares the pattern against identifiers in the source code and determines whether the custom rule is violated.


The tutorial uses the following 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);
}
```

User Interface (Desktop Products Only)

- 1 Create a Polyspace project. Add `printInitialValue.c` to the project.
- 2 On the **Configuration** pane, select **Coding Standards & Code Metrics**. Select the **Check custom rules** box.
- 3 Click .

The **Findings selection** window opens, displaying in the left pane all the coding standards Polyspace supports, and with the **Custom** node highlighted.

- 4 Specify the rules to check for in the right pane.

Expand the **4 Structs** node. For the option **4.3 All struct fields must follow the specified pattern**:

Column Title	Action
Status	Select <input checked="" type="checkbox"/> .
Convention	Enter All struct fields must begin with s_ and have capital letters or digits
Pattern	Enter s_[A-Z0-9_]+
Comment	Leave blank. This column is for comments that appear in the coding rules file alone.

- 5 Save the file and run the analysis. On the **Results List** pane, you see two violations of rule 4.3. Select the first violation.

- a On the **Source** pane, the line `int a;` is marked.
 - b On the **Result Details** pane, you see the error message that you had entered, All struct fields must begin with `s_` and have capital letters
- 6 Right-click the **Source** pane and select **Open Editor**. The file `printInitialValue.c` opens in the **Code Editor** pane or an external text editor depending on your **Preferences**.
- 7 In the file, replace all instances of `a` with `s_A` and `b` with `s_B`. Rerun the analysis.

The custom rule violations no longer appear on the **Results List** pane.

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. Make a writable copy of the file `custom_rules.xml` in `polyspaceserverroot\help\toolbox\polyspace_bug_finder_server\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\R2019a`.

For instance, for custom rule 4.3 to be disabled, the configuration file must contain these lines:

```
<standard name="CUSTOM RULES">
  ...
  <section name="4 Structs">
    ...
    <check id="4.3" state="off">
    </check>
    ...
  </section>
  ...
</standard>
```

Provide this file as 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 with Polyspace Code Prover Server, enter:

```
polyspace-code-prover-server -sources file -custom-rules from-file
                             -checkers-selection-file custom_rules.xml
```

See Also

Check custom rules (`-custom-rules`)

Compute Code Complexity Metrics

This example shows how to review the code complexity metrics that Polyspace computes. For information on the individual metrics, see “Code Metrics”.

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.

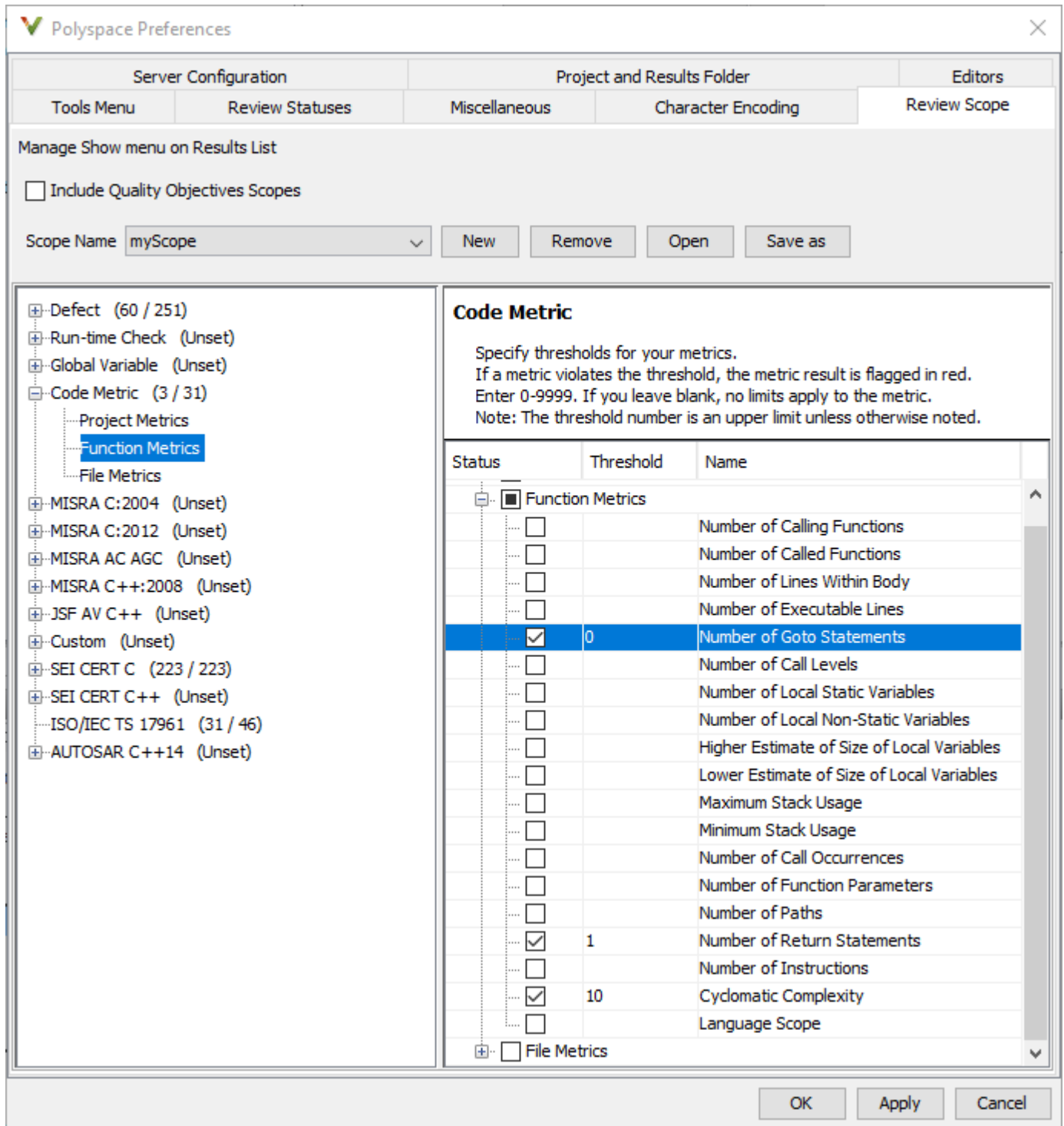
- 1 Select **Tools > Preferences**.
- 2 On the **Review Scope** tab, do one of the following:
 - To use a predefined limit, select **Include Quality Objectives Scopes**.

The **Scope Name** list shows the additional option **HIS**. The option **HIS** displays the **HIS** code metrics on page 12-52 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 appear on the **Results List** pane.
 - 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 12-52.

See Also

Calculate code metrics (-code-metrics)

More About

- “Code Metrics”
- “HIS Code Complexity Metrics” on page 12-52

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” on page 12-49.

Project

Polyspace evaluates the following HIS metrics at the project level.

Metric	Recommended Upper Limit
Number of direct recursions	0
Number of recursions	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	10
Language scope	4
Number of call levels	4
Number of calling functions	5
Number of called functions	7
Number of function parameters	5
Number of goto statements	0
Number of instructions	50
Number of paths	80
Number of return statements	1

See Also

More About

- “Compute Code Complexity Metrics” on page 12-49
- “Code Metrics”

Coding Rule Sets and Concepts

- “Polyspace MISRA C:2004 and MISRA AC AGC Checkers” on page 13-2
- “MISRA C:2004 and MISRA AC AGC Coding Rules” on page 13-3
- “Polyspace MISRA C:2012 Checkers” on page 13-38
- “Essential Types in MISRA C:2012 Rules 10.x” on page 13-39
- “Unsupported MISRA C:2012 Guidelines” on page 13-41
- “Polyspace MISRA C++ Checkers” on page 13-42
- “Unsupported MISRA C++ Coding Rules” on page 13-43
- “Polyspace JSF AV C++ Checkers” on page 13-47
- “JSF AV C++ Coding Rules” on page 13-48

Polyspace MISRA C:2004 and MISRA AC AGC Checkers

The Polyspace MISRA C:2004 checker helps you comply with the MISRA C 2004 coding standard.¹

When MISRA C rules are violated, the MISRA C checker enables Polyspace software to provide messages with information about the rule violations. Most messages are reported during the compile phase of an analysis.

The MISRA C checker can check nearly all of the **142** MISRA C:2004 rules.

The MISRA AC AGC checker checks rules from the OBL (obligatory) and REC (recommended) categories specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

There are subsets of MISRA coding rules that can have a direct or indirect impact on the selectivity (reliability percentage) of your results. When you set up rule checking, you can select these subsets directly. These subsets are defined in:

- “Software Quality Objective Subsets (C:2004)” on page 12-16
- “Software Quality Objective Subsets (AC AGC)” on page 12-20

Note The Polyspace MISRA checker is based on MISRA C:2004, which also incorporates MISRA C Technical Corrigendum.

See Also

More About

- “Check for Coding Standard Violations” on page 12-2
- “MISRA C:2004 and MISRA AC AGC Coding Rules” on page 13-3

1. MISRA and MISRA C are registered trademarks of MIRA Ltd., held on behalf of the MISRA Consortium.

MISRA C:2004 and MISRA AC AGC Coding Rules

In this section...

“Supported MISRA C:2004 and MISRA AC AGC Rules” on page 13-3

“Troubleshooting” on page 13-3

“List of Supported Coding Rules” on page 13-3

“Unsupported MISRA C:2004 and MISRA AC AGC Rules” on page 13-36

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 “Coding Standard Violations Not Displayed” on page 21-46.

List of Supported Coding Rules

- “Environment” on page 13-5
- “Language Extensions” on page 13-6
- “Documentation” on page 13-9
- “Character Sets” on page 13-9
- “Identifiers” on page 13-9
- “Types” on page 13-11
- “Constants” on page 13-11
- “Declarations and Definitions” on page 13-12

- “Initialisation” on page 13-15
- “Arithmetic Type Conversion” on page 13-16
- “Pointer Type Conversion” on page 13-19
- “Expressions” on page 13-20
- “Control Statement Expressions” on page 13-22
- “Control Flow” on page 13-25
- “Switch Statements” on page 13-27
- “Functions” on page 13-28
- “Pointers and Arrays” on page 13-29
- “Structures and Unions” on page 13-30
- “Preprocessing Directives” on page 13-30
- “Standard Libraries” on page 13-33
- “Runtime Failures” on page 13-36

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 <code>/** */</code> 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 <code>/*</code> shall not be used within a comment	The character sequence <code>/*</code> shall not appear within a comment.	This rule violation is also raised when the character sequence <code>/*</code> 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 Default Analysis" (Polyspace Bug Finder Access).</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 Default Analysis" (Polyspace Bug Finder Access).</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 Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
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 Default Analysis" (Polyspace Bug Finder Access).</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.
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.	
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. 	<p>Violations of this rule might be generated during the link phase.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p> <p><i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).</i></p>
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. 	<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
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.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
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.	<p>Restricted to explicit extern declarations (tentative definitions are ignored).</p> <p>Polyspace considers that variables or functions declared extern in a non-header file violate this rule.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p> <p><i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).</i></p>
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>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p> <p><i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).</i></p>

N.	MISRA Definition	Messages in report file	Polyspace Implementation
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 <code>main</code> function and you use options such as <code>Variables to initialize (-main-generator-writes-variables)</code> with value <code>custom</code> 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 <code>main</code> 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>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p> <p><i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).</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.		<p>Checked during code analysis.</p> <p>Violations displayed as Non-initialized variable results.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results. In Code Prover, you can also see a difference in results based on your choice for the option Verification level (-to). See "Check for Coding Standard Violations" on page 12-2.</p>
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.	<p>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.</p> <p>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:</p> <pre>int a = 2147483648;</pre> <p>There is a difference between decimal and hexadecimal constants when <code>int</code> and <code>long int</code> are not the same size.</p>

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.	<p>Casts and implicit conversions involving a function pointer.</p> <p>Casts or implicit conversions from <code>NULL</code> or <code>(void*)0</code> do not give any warning.</p>
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.	<p>There is also a warning on qualifier loss</p> <p>This rule maps to ISO/IEC TS 17961 ID <code>alignconv</code>.</p>

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, (var == 0).</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 flag is not a Boolean but an unsigned char.</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"> • [~/Left Shift/Right shift/&] operator applied on an expression whose underlying type is signed. • Bitwise ~ on operand of signed underlying type XX. • Bitwise [<< >>] on left hand operand of signed underlying type XX. • Bitwise [& ^] on two operands of s 	<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

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, check comparisons with at least one constant operand.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p> <ul style="list-style-type: none"> • Bug Finder flags some violations of this rule through the <code>Dead code</code> and <code>Useless if</code> checkers. • Code Prover does not use gray code to flag violations of this rule. <p>In Code Prover, you can also see a difference in results based on your choice for the option <code>Verification level (-to)</code>. See “Check for Coding Standard Violations” on page 12-2..</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 for 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;</pre>

N.	MISRA Definition	Messages in report file	Polyspace Implementation
			col++) {}

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.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
14.2	All non-null statements shall either have at least one side effect however executed, or cause control flow to change	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 	
14.3	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	We assume that a ';' is a null statement when it is the first character on a line (excluding comments). The rule is violated when: <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 <i>goto</i> statement shall not be used.	
14.5	The <i>continue</i> statement shall not be used.	The <i>continue</i> 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 <i>break</i> 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	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
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 <code>Number of Recursions</code> .
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.	
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.	
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	
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.	<p>Warning when assigning address to a global variable, returning a local variable address, or returning a parameter address.</p> <p>This rule maps to ISO/IEC TS 17961 ID accfree.</p>

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 "filename" sequence.	<ul style="list-style-type: none"> '<code>#include</code>' expects "FILENAME" or <code><FILENAME></code> '<code>#include_next</code>' expects "FILENAME" 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
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 : <code>sqrt</code>, <code>tan</code>, <code>pow</code>, <code>log</code>, <code>log10</code>, <code>fmod</code>, <code>acos</code>, <code>asin</code>, <code>acosh</code>, <code>atanh</code>, or <code>atan2</code>.</p> <p>Bug Finder and Code Prover check this rule differently. The analysis can produce different results.</p>
20.4	Dynamic heap memory allocation shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' 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 <code>errno</code> shall not be used	The error indicator <code>errno</code> shall not be used	Assumes that rule 20.2 is not violated

N.	MISRA Definition	Messages in report file	Polyspace Implementation
20.6	The macro <i>offsetof</i> , in library <code><stddef.h></code> , shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' 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 '<code><name></code>' shall not be used. • Identifier XX should not be used. 	In case the <i>longjmp</i> 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 <code><signal.h></code> shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' 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 <code><stdio.h></code> shall not be used in production code.	<ul style="list-style-type: none"> • The macro '<code><name></code>' 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 <i>atof</i> , <i>atoi</i> and <i>atoll</i> from library <code><stdlib.h></code> shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	In case the <i>atof</i> , <i>atoi</i> and <i>atoll</i> 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 <i>abort</i> , <i>exit</i> , <i>getenv</i> and <i>system</i> from library <code><stdlib.h></code> shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	In case the <i>abort</i> , <i>exit</i> , <i>getenv</i> and <i>system</i> functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.12	The time handling functions of library <code><time.h></code> shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' 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. 		Done by Polyspace. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results. In Code Prover, you can also see a difference in results based on your choice for the option <code>Verification level (-to)</code> . See "Check for Coding Standard Violations" on page 12-2..

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/compiler/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.

Polyspace MISRA C:2012 Checkers

The Polyspace MISRA C:2012 checker helps you to comply with the MISRA C 2012 coding standard.²

When MISRA C:2012 guidelines are violated, the Polyspace MISRA C:2012 checker provides messages with information about the violated rule or directive. Most violations are found during the compile phase of an analysis.

Polyspace Bug Finder can check all the MISRA C:2012 rules and most MISRA C:2012 directives. Polyspace Code Prover does not support checking of the following:

- MISRA C:2012 Dir 4.4, Dir 4.7, 4.13 and 4.14
- MISRA C:2012 Rule 21.13, 21.14, and 21.17 - 21.20
- MISRA C:2012 Rule 22.1 - 22.4 and 22.6 - 22.10

Each guideline is categorized into one of these three categories: mandatory, required, or advisory. When you set up rule checking, you can select subsets of these categories to check. For automatically generated code, some rules change categories, including to one additional category: readability. The `Use generated code requirements (-misra3-agc-mode)` option activates the categorization for automatically generated code.

There are additional subsets of MISRA C:2012 guidelines defined by Polyspace called Software Quality Objectives (SQQ) that can have a direct or indirect impact on the precision of your results. When you set up checking, you can select these subsets. These subsets are defined in “Software Quality Objective Subsets (C:2012)” on page 12-23.

See Also

Check MISRA C:2012 (-misra3) | Use generated code requirements (-misra3-agc-mode)

See Also

More About

- “Check for Coding Rule Violations” on page 5-19
- “MISRA C:2012 Directives and Rules”

2. MISRA and MISRA C are registered trademarks of MIRA Ltd., held on behalf of the MISRA Consortium.

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 5-19
- “MISRA C:2012 Directives and Rules”

Unsupported MISRA C:2012 Guidelines

The Polyspace coding rules checker does not check the following MISRA C:2012 directives. These directives are not checked either in Bug Finder or Code Prover. 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”

Polyspace MISRA C++ Checkers

The Polyspace MISRA C++ checker helps you comply with the MISRA C++:2008 coding standard.³

When MISRA C++ rules are violated, the Polyspace software provides messages with information about why the code violates the rule. Most violations are found during the compile phase of an analysis. The MISRA C++ checker can check 202 of the 230 MISRA C++ coding rules.

There are subsets of MISRA C++ coding rules that can have a direct or indirect impact on the selectivity (reliability percentage) of your results. When you set up rule checking, you can select these subsets directly. These subsets are defined in “Software Quality Objective Subsets (C++)” on page 12-26.

Note The Polyspace MISRA C++ checker is based on MISRA C++:2008 - “Guidelines for the use of the C++ language in critical systems.”

See Also

More About

- “Check for Coding Standard Violations” on page 12-2
- “MISRA C++:2008 Rules”

3. MISRA is a registered trademark of MIRA Ltd., held on behalf of the MISRA Consortium.

Unsupported MISRA C++ Coding Rules

In this section...
“Language Independent Issues” on page 13-43
“General” on page 13-44
“Lexical Conventions” on page 13-44
“Expressions” on page 13-44
“Declarations” on page 13-44
“Classes” on page 13-45
“Templates” on page 13-45
“Exception Handling” on page 13-45
“Library Introduction” on page 13-45

Polyspace does not check the following MISRAC++ coding rules. These rules are not checked either in Bug Finder or Code Prover. Some of these rules cannot be enforced because they are outside the scope of Polyspace software. The rules concern documentation, dynamic aspects, or functional aspects of MISRA rules.

For the list of supported rules, see “MISRA C++:2008 Rules”.

Language Independent Issues

N.	Category	MISRA Definition	Additional Information
0-1-4	Required	A project shall not contain non-volatile POD variables having only one use.	
0-1-6	Required	A project shall not contain instances of non-volatile variables being given values that are never subsequently used.	
0-1-8	Required	All functions with void return type shall have external side effects.	
0-3-1	Required	Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.	
0-3-2	Required	If a function generates error information, then that error information shall be tested.	
0-4-1	Document	Use of scaled-integer or fixed-point arithmetic shall be documented.	To observe this rule, check your compiler documentation.
0-4-2	Document	Use of floating-point arithmetic shall be documented.	To observe this rule, check your compiler documentation.

N.	Category	MISRA Definition	Additional Information
0-4-3	Document	Floating-point implementations shall comply with a defined floating-point standard.	To observe this rule, check your compiler documentation.

General

N.	Category	MISRA Definition	Additional Information
1-0-2	Document	Multiple compilers shall only be used if they have a common, defined interface.	To observe this rule, check your compiler documentation.
1-0-3	Document	The implementation of integer division in the chosen compiler shall be determined and documented.	To observe this rule, check your compiler documentation.

Lexical Conventions

N.	Category	MISRA Definition	Additional Information
2-2-1	Document	The character set and the corresponding encoding shall be documented.	To observe this rule, check your compiler documentation.

Expressions

N.	Category	MISRA Definition	Additional Information
5-0-16	Required	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	
5-17-1	Required	The semantic equivalence between a binary operator and its assignment operator form shall be preserved.	

Declarations

N.	Category	MISRA Definition	Additional Information
7-2-1	Required	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	
7-4-1	Document	All usage of assembler shall be documented.	To observe this rule, check your compiler documentation.

Classes

N.	Category	MISRA Definition	Additional Information
9-6-1	Document	When the absolute positioning of bits representing a bit-field is required, then the behavior and packing of bit-fields shall be documented.	To observe this rule, check your compiler documentation.

Templates

N.	Category	MISRA Definition	Additional Information
14-5-1	Required	A non-member generic function shall only be declared in a namespace that is not an associated namespace.	
14-7-1	Required	All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.	
14-7-2	Required	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.	

Exception Handling

N.	Category	MISRA Definition	Additional Information
15-0-1	Document	Exceptions shall only be used for error handling.	To observe this rule, check your compiler documentation.
15-1-1	Required	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	
15-3-1	Required	Exceptions shall be raised only after start-up and before termination of the program.	
15-3-4	Required	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.	

Library Introduction

N.	Category	MISRA Definition	Additional Information
17-0-3	Required	The names of standard library functions shall not be overridden.	

N.	Category	MISRA Definition	Additional Information
17-0-4	Required	All library code shall conform to MISRA C++.	To observe this rule, check your compiler documentation.

See Also

More About

- “MISRA C++:2008 Rules”

Polyspace JSF AV C++ Checkers

The Polyspace JSF C++ checker helps you comply with the Joint Strike Fighter® Air Vehicle C++ coding standards (JSF++). These coding standards were developed by Lockheed Martin® for the Joint Strike Fighter program. They are designed to improve the robustness of C++ code, and improve maintainability.

4

When JSF++ rules are violated, the Polyspace JSF C++ checker enables Polyspace software to provide messages with information about the rule violations. Most messages are reported during the compile phase of an analysis.

Note The Polyspace JSF C++ checker is based on JSF++:2005.

See Also

More About

- “Check for Coding Standard Violations” on page 12-2
- “JSF AV C++ Coding Rules” on page 13-48

4. JSF and Joint Strike Fighter are Lockheed Martin.

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.	<p>Rule is split into two parts: the definition of a macro function (29.def) and the call of a macrofunction (29.use).</p> <p>Messages in report file:</p> <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. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
41	Source lines will be kept to a length of 120 characters or less.	
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 if/else, do/while, for, and while 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 Default Analysis" (Polyspace Bug Finder Access).</i>

N.	JSF++ Definition	Polyspace Implementation
47	Identifiers will not begin with the underscore character '_'.	
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. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
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.

N.	JSF++ Definition	Polyspace Implementation
57	The public, protected, and private sections of a class will be declared in that order.	
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.	Messages in report file: <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.	Reports when the following characters are not directly connected to a white space: <ul style="list-style-type: none"> • . • -> • ! • ~ • - • ++ • — Note that a violation will be reported for "." used in float/double definition.

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 Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
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.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
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. <p>Only raw pointers are considered.</p> <ul style="list-style-type: none"> • 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.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

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 Default Analysis” (Polyspace Bug Finder Access).</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.	<p>Reports all duplicate declarations inside a translation unit. Reports when the declaration localization is not the same in all translation units.</p> <p><i>This checker is deactivated in a default Polyspace as You Code analysis. See “Checkers Deactivated in Polyspace as You Code Default Analysis” (Polyspace Bug Finder Access).</i></p>
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.	Done with Non-initialized variable checks in the software.
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. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
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.	<p>Reports user defined conversion function, non-explicit constructor with one parameter or default value for others (even undefined ones).</p> <p>Does not report copy-constructor.</p> <p>Additional message for constructor case:</p> <p>This constructor should be flagged as "explicit".</p>
178	<p>Down casting (casting from base to derived class) shall only be allowed through one of the following mechanism:</p> <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.	<p>Reports the following implicit casts :</p> <pre>integer => smaller integer unsigned => smaller or eq signed signed => smaller or eq un-signed integer => float float => integer</pre> <p>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></p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>
181	Redundant explicit casts will not be used.	Reports useless cast: <code>cast T to T</code> . Casts to equivalent <code>typedefs</code> 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. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
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.	<code>else if</code> should contain an <code>else</code> 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 <code>default</code> .
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 <code>loop variable</code> 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.
204	A single operation with side-effects shall only be used in the following contexts: <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 	Reports when: <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	The value of an expression shall be the same under any order of evaluation that the standard permits.	Reports when: <ul style="list-style-type: none"> • Variable is written more than once in an expression • Variable is read and write in sub-expressions • Volatile variable is accessed more than once <hr/> Note Read-write operations such as ++, are only considered as a write.
205	The volatile keyword shall not be used unless directly interfacing with hardware.	Reports if volatile keyword is used.

Memory Allocation

N.	JSF++ Definition	Polyspace Implementation
206	Allocation/deallocation from/to the free store (heap) shall not occur after initialization.	Reports calls to C library functions: malloc / calloc / realloc / free and all new/delete operators in functions or methods.

Fault Handling

N.	JSF++ Definition	Polyspace Implementation
208	C++ exceptions shall not be used.	Reports try, catch, throw spec, and throw.

Portable Code

N.	JSF++ Definition	Polyspace Implementation
209	The basic types of <code>int</code> , <code>short</code> , <code>long</code> , <code>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.	Only allows use of basic types through direct typedefs.
213	No dependence shall be placed on C++'s operator precedence rules, below arithmetic operators, in expressions.	<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>
215	Pointer arithmetic will not be used.	<p>Reports: <code>p + Ip - Ip++p--p+=p-=</code></p> <p>Allows <code>p[i]</code>.</p>

Unsupported JSF++ Rules

- “Code Size and Complexity” on page 13-67
- “Rules” on page 13-67
- “Environment” on page 13-67
- “Libraries” on page 13-67
- “Header Files” on page 13-67
- “Style” on page 13-68
- “Classes” on page 13-68
- “Namespaces” on page 13-69
- “Templates” on page 13-69
- “Functions” on page 13-69
- “Comments” on page 13-70
- “Initialization” on page 13-70
- “Types” on page 13-70
- “Unions and Bit Fields” on page 13-70
- “Operators” on page 13-70
- “Type Conversions” on page 13-71
- “Expressions” on page 13-71
- “Memory Allocation” on page 13-71
- “Portable Code” on page 13-71
- “Efficiency Considerations” on page 13-71
- “Miscellaneous” on page 13-71
- “Testing” on page 13-72

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)
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	<p>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.)</p> <p>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.</p>

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.

N.	JSF++ Definition
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.

N.	JSF++ Definition
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.
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.

N.	JSF++ Definition
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.

Configure Bug Finder Checkers

- “Choose Specific Bug Finder Defect Checkers” on page 14-2
- “Modify Default Behavior of Bug Finder Checkers” on page 14-4
- “Flag Deprecated or Unsafe Functions Using Bug Finder Checkers” on page 14-9
- “Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries” on page 14-11
- “Extend Bug Finder Checkers to Find Defects from Specific System Input Values” on page 14-13
- “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments” on page 14-16
- “Extend Checkers for Initialization to Check Function Arguments Passed by Pointers” on page 14-19
- “Prepare Checkers Configuration for Polyspace Bug Finder Analysis” on page 14-21
- “Short Names of Bug Finder Defect Checkers” on page 14-26
- “Bug Finder Defect Groups” on page 14-40
- “Sources of Tainting in a Polyspace Analysis” on page 14-45
- “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 14-48
- “Bug Finder Results Found in Fast Analysis Mode” on page 14-53
- “CWE Coding Standard and Polyspace Results” on page 14-78
- “Mapping Between CWE-658 or 659 and Polyspace Results” on page 14-103

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 14-48.
 - `all` for all defects.
 - `CWE` for defects related to CWE coding standard.

For more information, see “CWE Coding Standard and Polyspace Results” on page 14-78.
 - `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 Using Configuration Template” on page 1-16.

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 Checkers” on page 14-26.

See Also

`Find defects (-checkers)`

More About

- “Bug Finder Defect Groups” on page 14-40

- “Short Names of Bug Finder Defect Checkers” on page 14-26

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 14-2.

- 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

Option	Option Value	Checkers Modified	Modification
Find defects (-checkers)	Data race including atomic operations (user interface) or DATA_RACE_ALL (command line)	Data race	By default, the checker 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. See also “Define Atomic Operations in Multitasking Code” on page 11-24.

Option	Option Value	Checkers Modified	Modification
Run stricter checks considering all values of system inputs (-checks-using-system-input-values)		Checkers that rely on numerical values of system inputs	See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values” on page 14-13.
-code-behavior-specifications	XML file. Entries in the XML file map user-defined functions to functions from the Standard Library.	Checkers that detect issues with Standard Library functions	See “Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries” on page 14-11.
	XML file. Entries in the XML file map user-defined concurrency primitives to ones that Bug Finder can automatically detect.	Concurrency defects	See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments” on page 14-16.
	XML file. Entries in the XML file list functions that you want to prohibit from your source code.	Use of a forbidden function	See “Flag Deprecated or Unsafe Functions Using Bug Finder Checkers” on page 14-9.
	XML file. Entries in the XML file list functions whose pointer arguments must point to initialized buffers.	Non-initialized variable	See “Extend Checkers for Initialization to Check Function Arguments Passed by Pointers” on page 14-19.
-detect-bad-float-op-on-zero		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.

Option	Option Value	Checkers Modified	Modification
-consider-analysis-perimeter-as-trust-boundary		Tainted Data Defects	<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.

Coding Standard Checkers

Coding standards checkers can also be extended or modified with appropriate options.

Option	Option Value	Checkers Modified	Modification
Effective boolean types (-boolean-types)	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.
Allowed pragmas (-allowed-pragmas)	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.

Option	Option Value	Checkers Modified	Modification
-code-behavior-specifications	XML file. Entries in the XML file define limits on global aspects of your program such as maximum depth of nesting in control flow statements.	MISRA C: 2012 Rule 1.1	You can increase or decrease these parameters of the rule checker: <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
	XML file. Entries in the XML file define how many characters are compared before considering two identifiers as distinct.	MISRA C: 2012 Rules 5.1 to 5.5	These rules require uniqueness of certain types of identifiers. For instance, rule 5.1 requires that external identifiers be distinct. 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.
Check Guidelines (-guidelines)	Thresholds for software complexity checkers	Software Complexity	See “Reduce Software Complexity by Using Polyspace Checkers” on page 12-12

See Also

More About

- “Choose Specific Bug Finder Defect Checkers” on page 14-2
- “Bug Finder Defect Groups” on page 14-40

Flag Deprecated or Unsafe Functions Using Bug Finder Checkers

This topic shows how to create a custom list of functions and check for use of these functions 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. 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 blacklist 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';
}
```

Extend Checker

If the functions that you want to blacklist are not covered by the above checkers, use the checker `Use of a forbidden function`. To create a blacklist for the checker:

- 1 List functions 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 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 blacklist. 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 only checker that can be used to blacklist specified functions is the checker `Use of a forbidden function`.

See Also

`-code-behavior-specifications` | `Use of a forbidden function`

More About

- “Modify Default Behavior of Bug Finder Checkers” on page 14-4

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

See Also

`-code-behavior-specifications`

More About

- “Modify Default Behavior of Bug Finder Checkers” on page 14-4

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:

- Array access out of bounds

- 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
- Null pointer
- Shift of a negative value
- Shift operation overflow
- Sign change integer conversion overflow
- Unsigned integer conversion overflow
- Unsigned integer overflow
- Assertion

See Also

More About

- “Modify Default Behavior of Bug Finder Checkers” on page 14-4

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 11-5. 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 11-16.

Checkers That Can Be Extended

The concurrency defect checkers that can be extended in this way are:

- Data race
- Double lock and Double unlock
- Missing lock and Missing unlock
- Deadlock

See Also

-code-behavior-specifications

More About

- “Modify Default Behavior of Bug Finder Checkers” on page 14-4

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

The Non-initialized variable checker is extended using this option.

See Also

`-code-behavior-specifications`

More About

- “Modify Default Behavior of Bug Finder Checkers” on page 14-4

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 Coding Standard Violations” on page 12-2.

Coding Standards		
<input type="checkbox"/> Check MISRA C:2004	required-rules	View
<input type="checkbox"/> Check MISRA AC AGC	OBL-rules	View
<input type="checkbox"/> Check MISRA C:2012	mandatory-required	View
<input type="checkbox"/> Check SEI CERT-C	all	View
<input type="checkbox"/> Check ISO/IEC TS 17961	all	View
<input type="checkbox"/> Check custom rules	Edit	

- 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.

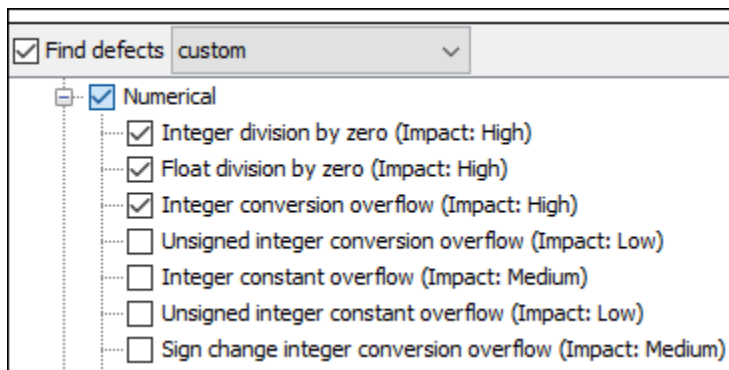
You might not be following standard coding practices in your organization and you might find external coding standards too sweeping for your preferences.

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 18-9.

- 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 14-2.

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 9-2.

- 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 Checkers” on page 14-26
- 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 Coding Standard Violations” on page 12-2.

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 2-16.

See Also

More About

- “Choose Specific Bug Finder Defect Checkers” on page 14-2
- “Check for Coding Standard Violations” on page 12-2

Short Names of Bug Finder Defect Checkers

To justify defects through code annotations, use the command-line names, or short names, listed in the following table.

You can also enable the detection of a specific defect by using its short name as argument of the `-checkers` option. Instead of listing individual defects, you can also specify groups of defects by the group name, for instance, `numerical`, `data_flow`, and so on. See `Find defects (-checkers)`.

Defect	Command-line Name
*this not returned in copy assignment operator	RETURN_NOT_REF_TO_THIS
A move operation may throw	MOVE_OPERATION_MAY_THROW
Abnormal termination of exit handler	EXIT_ABNORMAL_HANDLER
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
Ambiguous declaration syntax	MOST_VEXING_PARSE
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

Defect	Command-line Name
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
Const parameter values may cause unnecessary data copies	CONST_PARAMETER_VALUE
Const return values may cause unnecessary data copies	CONST_RETURN_VALUE
Const rvalue reference parameter may cause unnecessary data copies	CONST_RVALUE_REFERENCE_PARAMETER

Defect	Command-line Name
Const std::move input may cause a more expensive object copy	EXPENSIVE_STD_MOVE_CONST_OBJECT
Constant block cipher initialization vector	CRYPTO_CIPHER_CONSTANT_IV
Constant cipher key	CRYPTO_CIPHER_CONSTANT_KEY
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
Data race	DATA_RACE
Data race including atomic operations	DATA_RACE_ALL
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
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

Defect	Command-line Name
Empty destructors may cause unnecessary data copies	EMPTY_DESTRUCTOR_DEFINED
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 <code>c_str()</code> to <code>std::string</code> construction	EXPENSIVE_C_STR_STD_STRING_CONSTRUCTION
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 local variable copy	EXPENSIVE_LOCAL_VARIABLE
Expensive logical operation	EXPENSIVE_LOGICAL_OPERATION
Expensive pass by value	EXPENSIVE_PASS_BY_VALUE
Expensive return by value	EXPENSIVE_RETURN_BY_VALUE
Expensive use of non-member <code>std::string</code> operator+() instead of a simple append	EXPENSIVE_STD_STRING_APPEND
Expensive use of <code>std::string</code> methods instead of more efficient overload	EXPENSIVE_USE_OF_STD_STRING_METHODS
Expensive use of <code>std::string</code> with empty string literal	UNNECESSARY_EMPTY_STRING_LITERAL
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 <code>chroot</code> without <code>chdir</code>	CHROOT_MISUSE

Defect	Command-line Name
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
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
Incorrect key for cryptographic algorithm	CRYPTO_PKEY_INCORRECT_KEY
Incorrect order of network connection operations	BAD_NETWORK_CONNECT_ORDER

Defect	Command-line Name
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
Incorrect syntax of flexible array member size	FLEXIBLE_ARRAY_MEMBER_INCORRECT_SIZE
Incorrectly indented statement	INCORRECT_INDENTATION
Inefficient string length computation	INEFFICIENT_BASIC_STRING_LENGTH
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 use of = (assignment) operator	BAD_EQUAL_USE
Invalid use of == (equality) operator	BAD_EQUAL_EQUAL_USE
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
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
Library loaded from externally controlled path	TAINTED_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	TAINTED_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	TAINTED_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
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

Defect	Command-line Name
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
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

Defect	Command-line Name
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
Misuse of structure with flexible array member	FLEXIBLE_ARRAY_MEMBER_STRUCT_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
Multiple mutexes used with same conditional variable	MULTI_MUTEX_WITH_ONE_COND_VAR
Multiple threads waiting on same condition variable	SIGNALED_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

Defect	Command-line Name
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
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 invalid operation on boolean operand	INVALID_OPERATION_ON_BOOLEAN
Possible misuse of sizeof	SIZEOF_MISUSE
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
Qualifier removed in conversion	QUALIFIER_MISMATCH
Redundant expression in sizeof operand	SIZEOF_USELESS_OP

Defect	Command-line Name
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
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
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
Stream argument with possibly unintended side effects	STREAM_WITH_SIDE_EFFECT

Defect	Command-line Name
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
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
Unnamed namespace in header file	UNNAMED_NAMESPACE_IN_HEADER
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

Defect	Command-line Name
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
Use of a forbidden function	FORBIDDEN_FUNC
Useless if	USELESS_IF
Use of automatic variable as putenv-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 new or make_unique instead of more efficient make_shared	MISSING_MAKE_SHARED
Use of memset with size argument zero	MEMSET_INVALID_SIZE
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 tainted pointer	TAINTED_PTR

Defect	Command-line Name
Use of setjmp/longjmp	SETJMP_LONGJMP_USE
Use of undefined thread ID	UNDEFINED_THREAD_ID
Use of signal to kill thread	THREAD_KILLED_WITH_SIGNAL
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 17-6
- “Choose Specific Bug Finder Defect Checkers” on page 14-2

Bug Finder Defect Groups

In this section...

“C++ Exceptions” on page 14-40
 “Concurrency” on page 14-40
 “Cryptography” on page 14-41
 “Data flow” on page 14-41
 “Dynamic Memory” on page 14-42
 “Good Practice” on page 14-42
 “Numerical” on page 14-42
 “Object Oriented” on page 14-42
 “Performance” on page 14-43
 “Programming” on page 14-43
 “Resource Management” on page 14-43
 “Static Memory” on page 14-43
 “Security” on page 14-44
 “Tainted data” on page 14-44

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 18-2.

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

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 defects detect issues such as unnecessary data copies and inefficient C++ standard functions that can lead to performance bottlenecks in C++ code.

The defects include:

- `const` parameters or return values forcing copy instead of move operations
- Inefficient functions for newline insertion and string length computation

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`)

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 14-4

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”

See Also

`-consider-analysis-perimeter-as-trust-boundary` | `Find defects (-checkers)`

More About

- “Tainted data” on page 14-44
- “Modify Default Behavior of Bug Finder Checkers” on page 14-4

Polyspace Bug Finder Defects Checkers Enabled by Default

When you start a Bug Finder analysis, these checkers are enabled by default:

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
Deallocation of previously deallocated pointer	DOUBLE_DEALLOCATION

Defect	Command-line Name
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
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 = (assignment) operator	BAD_EQUAL_USE
Invalid use of == (equality) operator	BAD_EQUAL_EQUAL_USE
Invalid use of standard library floating point routine	FLOAT_STD_LIB
Invalid use of standard library integer routine	INT_STD_LIB

Defect	Command-line Name
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” (Polyspace Bug Finder Server).

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 == (equality) operator (bad_equal_equal_use)	Equality operation in assignment statement
Invalid use of = (assignment) 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.

CWE Coding Standard and Polyspace Results

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.

CWE and Polyspace Bug Finder

The CWE dictionary assigns a unique identifier to each software weakness type. These identifiers serve as a common language for describing software security weaknesses and a standard for software security tools targeting these weaknesses. For more information, see Common Weakness Enumeration.

Polyspace Bug Finder results can be mapped to CWE identifiers. Using Bug Finder, you can check and document if your software has weaknesses listed in the CWE dictionary. Bug Finder supports the following aspects of the CWE Compatibility and Effectiveness Program:

- **CWE Searchable:** For each supported CWE identifier, you can see all instances in your code that have weaknesses corresponding to the identifier.
- **CWE Output:** For each Polyspace Bug Finder defect:
 - You can view the associated CWE identifier.
 - You can report the associated CWE identifier.

Bug Finder results are mapped to CWE identifiers (IDs). Using the Bug Finder results, you can evaluate your code against the CWE standard. For instance, CWE ID 119 (Improper restriction of operations within the bounds of a memory buffer) maps to the Bug Finder defects, **Array access out of bounds** and **Pointer access out of bounds**.

For more information on the CWE Compatibility and Effectiveness Program, see CWE Compatibility.

Find CWE IDs from Polyspace Results

Use the following workflow if you want to focus your Bug Finder analysis on the CWE standard.

- *Analysis:* Check your code only for those Bug Finder defects that correspond to the standard. Use the option `Find defects (-checkers)` with value `CWE`.
- *Results:* If you enable only the defect checkers corresponding to the CWE standard, you see only the defects that correspond to the standard. Fix or justify each defect.

Along with defects, you can see the CWE IDs mapped to each defect in the **CWE ID** column on the **Results List** pane. If the column is not enabled by default, right-click any column header and select **CWE ID**.

- *Report:* When you generate a report, choose the `SecurityCWE` template tailored for the CWE standard. The report shows the CWE ID-s corresponding to each result.

Mapping Between CWE Identifiers and Polyspace Results

The following table lists the CWE IDs (version 3.3) addressed by Polyspace Bug Finder with its corresponding defect checkers. Using Polyspace Bug Finder defect checkers, you can check for 133 CWE IDs.

There are three types of CWE identifiers: Class, Base and Variant. Identifiers of type Class define security weaknesses at an abstract level independent of a specific language or technology, while identifiers of type Base and Variant are more concrete. On the other hand, Polyspace Bug Finder results are designed to be specific so that users can have a precise diagnosis of the defect in their code and understand the defect quickly. Therefore:

- The Bug Finder results are mapped to the specific identifiers of type Base and Variant rather than the generic identifiers of type Class.

Only when a result covers more ground than a specific CWE identifier is the result mapped to its more general parent type. For instance, the defect checker **Array access out of bounds** covers many kinds of buffer overflows, while CWE-788 refers only to “Access of Memory Location After End of Buffer”. Therefore, the defect checker is mapped to its parent, CWE-119, which refers to “Improper Restriction of Operations within the Bounds of a Memory Buffer”. However, to keep the mapping precise, an attempt is made to map to specific CWE identifiers.

- Often, more than one Bug Finder result is mapped to a certain CWE identifier.

For instance, CWE-908 refers to “Use of Uninitialized Resource”. To highlight specific kinds of uninitialized resources, Bug Finder has three different checkers: **Member not initialized in constructor**, **Non-initialized pointer**, and **Non-initialized variable**.

For mapping to the subsets CWE-658 and CWE-659, see “Mapping Between CWE-658 or 659 and Polyspace Results” on page 14-103.

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 Unsafe call to a system function
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

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
114	Process control	<p>Command executed from externally controlled path</p> <p>Execution of a binary from a relative path can be controlled by an external actor</p> <p>Execution of externally controlled command</p> <p>Library loaded from externally controlled path</p> <p>Load of library from a relative path can be controlled by an external actor</p>
119	Improper restriction of operations within the bounds of a memory buffer	<p>Array access out of bounds</p> <p>Pointer access out of bounds</p>
120	Buffer copy without checking size of input ('Classic buffer overflow')	<p>Invalid use of standard library memory routine</p> <p>Invalid use of standard library string routine</p> <p>Tainted NULL or non-null-terminated string</p>
121	Stack-based buffer overflow	<p>Array access with tainted index</p> <p>Destination buffer overflow in string manipulation</p>
122	Heap-based buffer overflow	<p>Pointer dereference with tainted offset</p>
124	Buffer underwrite ('Buffer underflow')	<p>Array access with tainted index</p> <p>Buffer overflow from incorrect string format specifier</p> <p>Destination buffer underflow in string manipulation</p> <p>Pointer dereference with tainted offset</p>
125	Out-of-bounds read	<p>Array access with tainted index</p> <p>Buffer overflow from incorrect string format specifier</p> <p>Destination buffer overflow in string manipulation</p>
126	Buffer over-read	<p>Buffer overflow from incorrect string format specifier</p>

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
127	Buffer under-read	Buffer overflow from incorrect string format specifier
128	Wrap-around error	Integer constant overflow Integer conversion overflow Integer overflow Memory allocation with tainted size Tainted sign change conversion Tainted size of variable length array Unsigned integer constant overflow Unsigned integer conversion overflow Unsigned integer overflow
129	Improper validation of array index	Array access with tainted index Pointer dereference with tainted offset
130	Improper handling of length parameter inconsistency	Mismatch between data length and size
131	Incorrect calculation of buffer size	Array access out of bounds Memory allocation with tainted size Pointer access out of bounds Tainted sign change conversion Tainted size of variable length array Unsigned integer conversion overflow Unsigned integer overflow
134	Uncontrolled format string	Tainted string format
135	Incorrect Calculation of Multi-Byte String Length	Destination buffer overflow in string manipulation Misuse of narrow or wide character string Unreliable cast of pointer

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
170	Improper null termination	Missing null in string array Misuse of readlink() Tainted NULL or non-null-terminated string
188	Reliance on data/memory layout	Invalid assumptions about memory organization Memory comparison of padding data Memory comparison of strings Missing byte reordering when transferring data Pointer access out of bounds
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 Shift of a negative value Shift operation overflow Tainted division operand Unsigned integer constant overflow Unsigned integer conversion overflow Unsigned integer overflow

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
190	Integer overflow or wraparound	Integer conversion overflow Integer constant overflow Integer overflow Integer precision exceeded Possible invalid operation on boolean operand Shift operation overflow Tainted division operand Unsigned integer conversion overflow Unsigned integer overflow Unsigned integer constant overflow
191	Integer underflow (Wrap or wraparound)	Integer constant overflow Integer conversion overflow Integer overflow Unsigned integer constant overflow Unsigned integer conversion overflow Unsigned integer overflow
192	Integer coercion error	Integer conversion overflow Integer overflow Sign change integer conversion overflow Tainted sign change conversion Unsigned integer conversion overflow Unsigned integer overflow
194	Unexpected sign extension	Sign change integer conversion overflow Tainted sign change conversion
195	Signed to unsigned conversion error	Sign change integer conversion overflow Tainted sign change conversion
196	Unsigned to signed conversion error	Sign change integer conversion overflow

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
197	Numeric truncation error	Float conversion overflow Integer conversion overflow Unsigned integer conversion overflow
198		Missing byte reordering when transferring data
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
242	Use of inherently dangerous function	Use of dangerous standard function
243	Creation of chroot jail without changing working directory	File manipulation after chroot without chdir
244	Improper clearing of heap memory before release	Sensitive heap memory not cleared before release
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
252	Unchecked return value	Returned value of a sensitive function not checked

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
253	Incorrect Check of Function Return Value	Errno not checked Errno not reset Returned value of a sensitive function not checked Unprotected dynamic memory allocation Unsafe conversion from string to numerical value
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
312	Cleartext Storage of Sensitive Information	Sensitive heap memory not cleared before release Uncleared sensitive data in stack
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

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
325	Missing required cryptographic step	Context initialized incorrectly for cryptographic operation Incorrect key for cryptographic algorithm Missing block cipher initialization vector Missing cipher data to process Missing cipher final step Missing cipher algorithm Missing cipher key Missing data for encryption, decryption or signing operation Missing parameters for key generation No data added into context Weak cipher algorithm Weak cipher mode
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

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
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
330	Use of insufficiently random values	Deterministic random output from constant seed Predictable block cipher initialization vector Predictable cipher key Predictable random output from predictable seed Vulnerable pseudo-random number generator
336	Same seed in PRNG	Deterministic random output from constant seed
337	Predictable seed in PRNG	Predictable random output from predictable seed
338	Use of cryptographically weak pseudo-random number generator (PRNG)	Predictable block cipher initialization vector Predictable cipher key Vulnerable pseudo-random number generator

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
353	Missing Support for Integrity Check	Context initialized incorrectly for digest operation Nonsecure hash algorithm
354	Improper Validation of Integrity Check Value	Context initialized incorrectly for digest operation
362	Concurrent execution using shared resource with improper synchronization ('Race Condition')	File descriptor exposure to child process Opening previously opened resource
364	Signal handler race condition	Function called from signal handler not asynchronous-safe (strict) Function called from signal handler not asynchronous-safe Shared data access within signal handler
366	Race condition within a thread	Data race including atomic operations Data race through standard library function call Data race
367	Time-of-check time-of-use (TOCTOU) race condition	File access between time of check and use (TOCTOU)
369	Divide by zero	Float division by zero Integer division by zero Invalid use of standard library floating point routine Invalid use of standard library integer routine Tainted division operand Tainted modulo operand

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
372	Incomplete internal state distinction	Context initialized incorrectly for cryptographic operation Context initialized incorrectly for digest operation Incompatible padding for RSA algorithm operation Inconsistent cipher operations Missing cipher data to process Missing cipher final step Missing data for encryption, decryption or signing operation Missing parameters for key generation
375	Returning a mutable object to an untrusted caller	Return of non const handle to encapsulated data member
377	Insecure temporary file	Use of non-secure temporary file
387	Signal errors	Function called from signal handler not asynchronous-safe (strict) Function called from signal handler not asynchronous-safe 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
401	Improper release of memory before removing last reference	Memory leak Thread-specific memory leak
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

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
413	Improper Resource Locking	Data race Data race including atomic operations Data race through standard library function call Function called from signal handler not asynchronous-safe Function called from signal handler not asynchronous-safe (strict) Opening previously opened resource Shared data access within signal handler
415	Double free	Deallocation of previously deallocated pointer Missing reset of a freed pointer
416	Use after free	Missing reset of a freed pointer Use of previously freed pointer
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 Library loaded from externally controlled path Load of library from a relative path can be controlled by an external actor Use of externally controlled environment variable
456	Missing initialization of a variable	Errno not reset Member not initialized in constructor Non-initialized pointer Non-initialized variable
457	Use of uninitialized variable	Member not initialized in constructor Non-initialized pointer Non-initialized variable

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
465	Pointer Issues	Unsafe conversion between pointer and integer
466	Return of pointer value outside of expected range	Array access out of bounds Pointer access out of bounds Unsafe conversion between pointer and integer
467	Use of sizeof() on a pointer type	Possible misuse of sizeof Wrong type used in sizeof
468	Incorrect pointer scaling	Incorrect pointer scaling
469	Use of pointer subtraction to determine size	Subtraction or comparison between pointers to different arrays
471	Modification of assumed-immutable data	Writing to const qualified object
474	Use of function with inconsistent implementations	Signal call from within signal handler Use of obsolete standard function
475	Undefined behavior for input to API	Copy of overlapping memory
476	NULL pointer dereference	Null pointer Tainted NULL or non-null-terminated string
477	Use of obsolete functions	Use of obsolete standard function
478	Missing default case in switch statement	Missing case for switch condition
479	Signal handler use of a non-reentrant function	Function called from signal handler not asynchronous-safe (strict) Function called from signal handler not asynchronous-safe
480	Use of incorrect operator	Invalid use of = (assignment) operator Invalid use of == (equality) operator
481	Assigning instead of comparing	Invalid use of = (assignment) operator
482	Comparing instead of assigning	Invalid use of == (equality) operator
483	Incorrect block delimitation	Incorrectly indented statement Semicolon on same line as if, for or while statement

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
484	Omitted break statement in switch	Missing break of switch case
522	Insufficiently Protected Credentials	Constant cipher key Nonsecure hash algorithm Nonsecure parameters for key generation Nonsecure RSA public exponent Nonsecure SSL/TLS protocol Unsafe standard encryption function
532	Information exposure through log files	Sensitive data printed out
534	Information exposure through debug log files	Sensitive data printed out
535	Information exposure through shell error message	Sensitive data printed out
547	Use of hard-coded, security-relevant constants	Hard coded buffer size Hard coded loop boundary
558	Use of getlogin() in multithreaded application	Unsafe standard function
560	Use of umask() with chmod-style argument	Umask used with chmod-style arguments
561	Dead code	Dead code Static uncalled function Unreachable code
562	Return of stack variable address	Pointer or reference to stack variable leaving scope Use of automatic variable as putenv-family function argument

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
573	Improper following of specification by caller	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 cipher data to process Missing cipher final step Missing cipher algorithm Missing cipher key Missing data for encryption, decryption or signing operation Missing final step after hashing update operation Missing hash algorithm Missing parameters for key generation Missing peer key Missing private key for X.509 certificate Missing private key Missing public key Modification of internal buffer returned from nonreentrant standard function TLS/SSL connection method not set TLS/SSL connection method set incorrectly
587	Assignment of a fixed address to a pointer	Function pointer assigned with absolute address Unsafe conversion between pointer and integer
590	Free of memory not on the heap	Invalid free of pointer

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
606	Unchecked input for loop condition	Loop bounded with tainted value
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 nonreentrant standard function Standard function call with incorrect arguments
658	See “Mapping Between CWE-658 or 659 and Polyspace Results” on page 14-103.	
659	See “Mapping Between CWE-658 or 659 and Polyspace Results” on page 14-103.	
663	Use of a non-reentrant function in a concurrent context	Function called from signal handler not asynchronous-safe (strict) Function called from signal handler not asynchronous-safe 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

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
665	Improper initialization	Call to memset with unintended value Improper array initialization Overlapping assignment Use of memset with size argument zero
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 a previously closed resource Use of previously closed resource
675	Duplicate operations on resource	Opening previously opened resource
676	Use of potentially dangerous function	Unsafe conversion from string to numerical value Use of dangerous standard function
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 numerical value
683	Function Call With Incorrect Order of Arguments	Call to memset with unintended value Format string specifiers and arguments mismatch

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
685	Function call with incorrect number of arguments	Declaration mismatch Format string specifiers and arguments mismatch Standard function call with incorrect arguments Too many va_arg calls for current argument list
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 nonpositive size
690	Unchecked return value to null pointer dereference	Invalid use of standard library memory routine Invalid use of standard library routine Invalid use of standard library string routine Null pointer Returned value of a sensitive function not checked Standard function call with incorrect arguments Tainted NULL or non-null-terminated string Unprotected dynamic memory allocation

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
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
704	Incorrect type conversion or cast	Character value absorbed into EOF Misuse of sign-extended character value Precision loss in integer to float conversion Qualifier removed in conversion Unreliable cast of pointer Wrong allocated object size for cast
705	Incorrect control flow scoping	Abnormal termination of exit handler
710	Coding standard violation	Bitwise and arithmetic operation 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
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 numerical value
759	Use of a One-Way Hash without a Salt	Missing salt for hashing operation
762	Mismatched memory management routines	Invalid free of pointer Mismatched alloc/dealloc functions on Windows
764	Multiple locks of a critical resource	Double lock
765	Multiple unlocks of a critical resource	Double unlock

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
767	Access to critical private variable via public method	Return of non const handle to encapsulated data member
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
783	Operator precedence logic error	Possibly unintended evaluation of expression because of operator precedence rules
785	Use of path manipulation function without maximum-sized buffer	Use of path manipulation function without maximum sized buffer checking
786	Access of memory location before start of buffer	Destination buffer underflow in string manipulation
787	Out-of-bounds write	Destination buffer overflow in string manipulation Destination buffer underflow in string manipulation
789	Uncontrolled memory allocation	Memory allocation with tainted size Tainted size of variable length array Unprotected dynamic memory allocation
805	Buffer access with incorrect length value	Hard-coded object size used to manipulate memory
822	Untrusted pointer dereference	Tainted NULL or non-null-terminated string
823	Use of out-of-range pointer offset	Pointer access out of bounds Pointer dereference with tainted offset
824	Access of uninitialized pointer	Non-initialized pointer

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
825	Expired Pointer Dereference	Accessing object with temporary lifetime Deallocation of previously deallocated pointer Environment pointer invalidated by previous operation Missing reset of a freed pointer Pointer or reference to stack variable leaving scope Use of automatic variable as putenv-family function argument Use of previously freed pointer
826	Premature release of resource during expected lifetime	Closing a 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 (strict) Function called from signal handler not asynchronous-safe
832	Unlock of a resource that is not locked	Missing lock
833	Deadlock	Deadlock
843	Access of resource using incompatible type ('Type confusion')	Unreliable cast of pointer
872	CERT C++ Secure Coding Section 04 - Integers (INT)	Invalid use of standard library integer routine
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

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
908	Use of uninitialized resource	Member not initialized in constructor Non-initialized pointer Non-initialized variable
910	Use of expired file descriptor	Closing a previously closed resource Standard function call with incorrect arguments Use of previously closed resource
922	Insecure Storage of Sensitive Information	File manipulation after chroot without chdir Umask used with chmod-style arguments Use of non-secure temporary file Vulnerable permission assignments

Mapping Between CWE-658 or 659 and Polyspace Results

CWE-658: Weaknesses in Software Written in C

CWE-658 is a subset of CWE IDs found in C programs that are not common to all languages. See CWE-658.

The following table lists the CWE IDs (version 3.3) from this subset that are addressed by Polyspace Bug Finder, with its corresponding defect checkers.

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
119	Improper restriction of operations within the bounds of a memory buffer	Array access out of bounds Pointer access out of bounds
120	Buffer copy without checking size of input ('Classic buffer overflow')	Invalid use of standard library memory routine Invalid use of standard library string routine Tainted NULL or non-null-terminated string
121	Stack-based buffer overflow	Array access with tainted index Destination buffer overflow in string manipulation
122	Heap-based buffer overflow	Pointer dereference with tainted offset
124	Buffer underwrite ('Buffer underflow')	Array access with tainted index Buffer overflow from incorrect string format specifier Destination buffer underflow in string manipulation Pointer dereference with tainted offset
125	Out-of-bounds read	Array access with tainted index Buffer overflow from incorrect string format specifier Destination buffer overflow in string manipulation
126	Buffer over-read	Buffer overflow from incorrect string format specifier
127	Buffer under-read	Buffer overflow from incorrect string format specifier

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
128	Wrap-around error	Integer constant overflow Integer conversion overflow Integer overflow Memory allocation with tainted size Tainted sign change conversion Tainted size of variable length array Unsigned integer constant overflow Unsigned integer conversion overflow Unsigned integer overflow
129	Improper validation of array index	Array access with tainted index Pointer dereference with tainted offset
130	Improper handling of length parameter inconsistency	Mismatch between data length and size
131	Incorrect calculation of buffer size	Array access out of bounds Memory allocation with tainted size Pointer access out of bounds Tainted sign change conversion Tainted size of variable length array Unsigned integer conversion overflow Unsigned integer overflow
134	Uncontrolled format string	Tainted string format
135	Incorrect Calculation of Multi-Byte String Length	Destination buffer overflow in string manipulation Misuse of narrow or wide character string Unreliable cast of pointer
170	Improper null termination	Missing null in string array Misuse of readlink() Tainted NULL or non-null-terminated string

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
188	Reliance on data/ memory layout	Invalid assumptions about memory organization Memory comparison of padding data Memory comparison of strings Missing byte reordering when transferring data Pointer access out of bounds
191	Integer underflow (Wrap or wraparound)	Integer constant overflow Integer conversion overflow Integer overflow Unsigned integer constant overflow Unsigned integer conversion overflow Unsigned integer overflow
192	Integer coercion error	Integer conversion overflow Integer overflow Sign change integer conversion overflow Tainted sign change conversion Unsigned integer conversion overflow Unsigned integer overflow
194	Unexpected sign extension	Sign change integer conversion overflow Tainted sign change conversion
195	Signed to unsigned conversion error	Sign change integer conversion overflow Tainted sign change conversion
196	Unsigned to signed conversion error	Sign change integer conversion overflow
197	Numeric truncation error	Float conversion overflow Integer conversion overflow Unsigned integer conversion overflow
242	Use of inherently dangerous function	Use of dangerous standard function
243	Creation of chroot jail without changing working directory	File manipulation after chroot without chdir

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
244	Improper clearing of heap memory before release	Sensitive heap memory not cleared before release
362	Concurrent execution using shared resource with improper synchronization ('Race Condition')	File descriptor exposure to child process Opening previously opened resource
364	Signal handler race condition	Function called from signal handler not asynchronous-safe (strict) Function called from signal handler not asynchronous-safe Shared data access within signal handler
366	Race condition within a thread	Data race including atomic operations Data race through standard library function call Data race
375	Returning a mutable object to an untrusted caller	Return of non const handle to encapsulated data member
401	Improper release of memory before removing last reference	Memory leak Thread-specific memory leak
415	Double free	Deallocation of previously deallocated pointer Missing reset of a freed pointer
416	Use after free	Missing reset of a freed pointer Use of previously freed pointer
457	Use of uninitialized variable	Member not initialized in constructor Non-initialized pointer Non-initialized variable
466	Return of pointer value outside of expected range	Array access out of bounds Pointer access out of bounds Unsafe conversion between pointer and integer
467	Use of sizeof() on a pointer type	Possible misuse of sizeof Wrong type used in sizeof

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
468	Incorrect pointer scaling	Incorrect pointer scaling
469	Use of pointer subtraction to determine size	Subtraction or comparison between pointers to different arrays
474	Use of function with inconsistent implementations	Signal call from within signal handler Use of obsolete standard function
476	NULL pointer dereference	Null pointer Tainted NULL or non-null-terminated string
478	Missing default case in switch statement	Missing case for switch condition
479	Signal handler use of a non-reentrant function	Function called from signal handler not asynchronous-safe (strict) Function called from signal handler not asynchronous-safe
480	Use of incorrect operator	Invalid use of = (assignment) operator Invalid use of == (equality) operator
481	Assigning instead of comparing	Invalid use of = (assignment) operator
482	Comparing instead of assigning	Invalid use of == (equality) operator
483	Incorrect block delimitation	Incorrectly indented statement Semicolon on same line as if, for or while statement
484	Omitted break statement in switch	Missing break of switch case
558	Use of getlogin() in multithreaded application	Unsafe standard function
560	Use of umask() with chmod-style argument	Umask used with chmod-style arguments
562	Return of stack variable address	Pointer or reference to stack variable leaving scope Use of automatic variable as putenv-family function argument
587	Assignment of a fixed address to a pointer	Function pointer assigned with absolute address Unsafe conversion between pointer and integer

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
676	Use of potentially dangerous function	Unsafe conversion from string to numerical value Use of dangerous standard function
685	Function call with incorrect number of arguments	Declaration mismatch Format string specifiers and arguments mismatch Standard function call with incorrect arguments Too many va_arg calls for current argument list
690	Unchecked return value to null pointer dereference	Invalid use of standard library memory routine Invalid use of standard library routine Invalid use of standard library string routine Null pointer Returned value of a sensitive function not checked Standard function call with incorrect arguments Tainted NULL or non-null-terminated string Unprotected dynamic memory allocation
704	Incorrect type conversion or cast	Character value absorbed into EOF Misuse of sign-extended character value Precision loss in integer to float conversion Qualifier removed in conversion Unreliable cast of pointer Wrong allocated object size for cast
762	Mismatched memory management routines	Invalid free of pointer Mismatched alloc/dealloc functions on Windows
783	Operator precedence logic error	Possibly unintended evaluation of expression because of operator precedence rules

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
785	Use of path manipulation function without maximum-sized buffer	Use of path manipulation function without maximum sized buffer checking
787	Out-of-bounds write	Destination buffer overflow in string manipulation Destination buffer underflow in string manipulation
789	Uncontrolled memory allocation	Memory allocation with tainted size Tainted size of variable length array Unprotected dynamic memory allocation
805	Buffer access with incorrect length value	Hard-coded object size used to manipulate memory
843	Access of resource using incompatible type ('Type confusion')	Unreliable cast of pointer
910	Use of expired file descriptor	Closing a previously closed resource Standard function call with incorrect arguments Use of previously closed resource

CWE-659: Weaknesses in Software Written in C++

CWE-659 is a subset of CWE IDs found in C++ programs that are not common to all languages. See CWE-659.

The following table lists the CWE IDs (version 3.3) from this subset that are addressed by Polyspace Bug Finder, with its corresponding defect checkers.

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
119	Improper restriction of operations within the bounds of a memory buffer	Array access out of bounds Pointer access out of bounds
120	Buffer copy without checking size of input ('Classic buffer overflow')	Invalid use of standard library memory routine Invalid use of standard library string routine Tainted NULL or non-null-terminated string

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
121	Stack-based buffer overflow	Array access with tainted index Destination buffer overflow in string manipulation
122	Heap-based buffer overflow	Pointer dereference with tainted offset
124	Buffer underwrite ('Buffer underflow')	Array access with tainted index Buffer overflow from incorrect string format specifier Destination buffer underflow in string manipulation Pointer dereference with tainted offset
125	Out-of-bounds read	Array access with tainted index Buffer overflow from incorrect string format specifier Destination buffer overflow in string manipulation
126	Buffer over-read	Buffer overflow from incorrect string format specifier
127	Buffer under-read	Buffer overflow from incorrect string format specifier
128	Wrap-around error	Integer constant overflow Integer conversion overflow Integer overflow Memory allocation with tainted size Tainted sign change conversion Tainted size of variable length array Unsigned integer constant overflow Unsigned integer conversion overflow Unsigned integer overflow
129	Improper validation of array index	Array access with tainted index Pointer dereference with tainted offset
130	Improper handling of length parameter inconsistency	Mismatch between data length and size

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
131	Incorrect calculation of buffer size	<p>Array access out of bounds</p> <p>Memory allocation with tainted size</p> <p>Pointer access out of bounds</p> <p>Tainted sign change conversion</p> <p>Tainted size of variable length array</p> <p>Unsigned integer conversion overflow</p> <p>Unsigned integer overflow</p>
134	Uncontrolled format string	Tainted string format
135	Incorrect Calculation of Multi-Byte String Length	<p>Destination buffer overflow in string manipulation</p> <p>Misuse of narrow or wide character string</p> <p>Unreliable cast of pointer</p>
170	Improper null termination	<p>Missing null in string array</p> <p>Misuse of readlink()</p> <p>Tainted NULL or non-null-terminated string</p>
188	Reliance on data/memory layout	<p>Invalid assumptions about memory organization</p> <p>Memory comparison of padding data</p> <p>Memory comparison of strings</p> <p>Missing byte reordering when transferring data</p> <p>Pointer access out of bounds</p>
191	Integer underflow (Wrap or wraparound)	<p>Integer constant overflow</p> <p>Integer conversion overflow</p> <p>Integer overflow</p> <p>Unsigned integer constant overflow</p> <p>Unsigned integer conversion overflow</p> <p>Unsigned integer overflow</p>

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
192	Integer coercion error	Integer conversion overflow Integer overflow Sign change integer conversion overflow Tainted sign change conversion Unsigned integer conversion overflow Unsigned integer overflow
194	Unexpected sign extension	Sign change integer conversion overflow Tainted sign change conversion
195	Signed to unsigned conversion error	Sign change integer conversion overflow Tainted sign change conversion
196	Unsigned to signed conversion error	Sign change integer conversion overflow
197	Numeric truncation error	Float conversion overflow Integer conversion overflow Unsigned integer conversion overflow
242	Use of inherently dangerous function	Use of dangerous standard function
243	Creation of chroot jail without changing working directory	File manipulation after chroot without chdir
244	Improper clearing of heap memory before release	Sensitive heap memory not cleared before release
362	Concurrent execution using shared resource with improper synchronization ('Race Condition')	File descriptor exposure to child process Opening previously opened resource
364	Signal handler race condition	Function called from signal handler not asynchronous-safe (strict) Function called from signal handler not asynchronous-safe Shared data access within signal handler

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
366	Race condition within a thread	Data race including atomic operations Data race through standard library function call Data race
375	Returning a mutable object to an untrusted caller	Return of non const handle to encapsulated data member
401	Improper release of memory before removing last reference	Memory leak Thread-specific memory leak
415	Double free	Deallocation of previously deallocated pointer Missing reset of a freed pointer
416	Use after free	Missing reset of a freed pointer Use of previously freed pointer
457	Use of uninitialized variable	Member not initialized in constructor Non-initialized pointer Non-initialized variable
466	Return of pointer value outside of expected range	Array access out of bounds Pointer access out of bounds Unsafe conversion between pointer and integer
467	Use of sizeof() on a pointer type	Possible misuse of sizeof Wrong type used in sizeof
468	Incorrect pointer scaling	Incorrect pointer scaling
469	Use of pointer subtraction to determine size	Subtraction or comparison between pointers to different arrays
476	NULL pointer dereference	Null pointer Tainted NULL or non-null-terminated string
478	Missing default case in switch statement	Missing case for switch condition
479	Signal handler use of a non-reentrant function	Function called from signal handler not asynchronous-safe (strict) Function called from signal handler not asynchronous-safe

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
480	Use of incorrect operator	Invalid use of = (assignment) operator Invalid use of == (equality) operator
481	Assigning instead of comparing	Invalid use of = (assignment) operator
482	Comparing instead of assigning	Invalid use of == (equality) operator
483	Incorrect block delimitation	Incorrectly indented statement Semicolon on same line as if, for or while statement
484	Omitted break statement in switch	Missing break of switch case
558	Use of getlogin() in multithreaded application	Unsafe standard function
562	Return of stack variable address	Pointer or reference to stack variable leaving scope Use of automatic variable as putenv-family function argument
587	Assignment of a fixed address to a pointer	Function pointer assigned with absolute address Unsafe conversion between pointer and integer
676	Use of potentially dangerous function	Unsafe conversion from string to numerical value Use of dangerous standard function
690	Unchecked return value to null pointer dereference	Invalid use of standard library memory routine Invalid use of standard library routine Invalid use of standard library string routine Null pointer Returned value of a sensitive function not checked Standard function call with incorrect arguments Tainted NULL or non-null-terminated string Unprotected dynamic memory allocation

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
704	Incorrect type conversion or cast	Character value absorbed into EOF Misuse of sign-extended character value Precision loss in integer to float conversion Qualifier removed in conversion Unreliable cast of pointer Wrong allocated object size for cast
762	Mismatched memory management routines	Invalid free of pointer Mismatched alloc/dealloc functions on Windows
767	Access to critical private variable via public method	Return of non const handle to encapsulated data member
783	Operator precedence logic error	Possibly unintended evaluation of expression because of operator precedence rules
785	Use of path manipulation function without maximum-sized buffer	Use of path manipulation function without maximum sized buffer checking
787	Out-of-bounds write	Destination buffer overflow in string manipulation Destination buffer underflow in string manipulation
789	Uncontrolled memory allocation	Memory allocation with tainted size Tainted size of variable length array Unprotected dynamic memory allocation
805	Buffer access with incorrect length value	Hard-coded object size used to manipulate memory
843	Access of resource using incompatible type ('Type confusion')	Unreliable cast of pointer
910	Use of expired file descriptor	Closing a previously closed resource Standard function call with incorrect arguments Use of previously closed resource

See Also

More About


- “CWE Coding Standard and Polyspace Results” on page 14-78

Configure Comment Import from Previous Results

- “Import Review Information from Previous Polyspace Analysis” on page 15-2
- “Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results” on page 15-6

Import Review Information from Previous Polyspace Analysis

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 Polyspace Results Through Bug Fixes or Justifications” on page 17-2.

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 upload results to the Polyspace Access web interface, review information from the last run of the same project are applied to the current run. You cannot disable the automatic import.

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 15-3.

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 `numRecipients` 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:

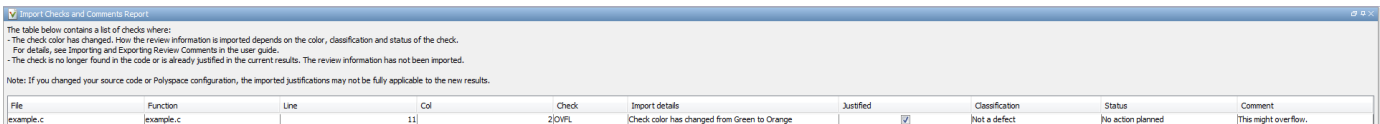
- When justifying nonlocal results that are related to previous events, use careful judgement.
- For critical components, conduct periodic assessments of justified results to see if the justifications still apply. Such assessments are useful specially for the Code Prover run-time checks.

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	12	2	2(DIV)	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

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 15-2.

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-statement...	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 17-6

Interpret Polyspace Bug Finder Results


- “Interpret Bug Finder Results in Polyspace Desktop User Interface” on page 16-2
- “Investigate the Cause of Empty Results List” on page 16-7
- “Dashboard” on page 16-9
- “Concurrency Modeling” on page 16-13
- “Results List” on page 16-15
- “Source” on page 16-18
- “Call Hierarchy” on page 16-23
- “Result Details” on page 16-25

Interpret Bug Finder Results in Polyspace Desktop User Interface

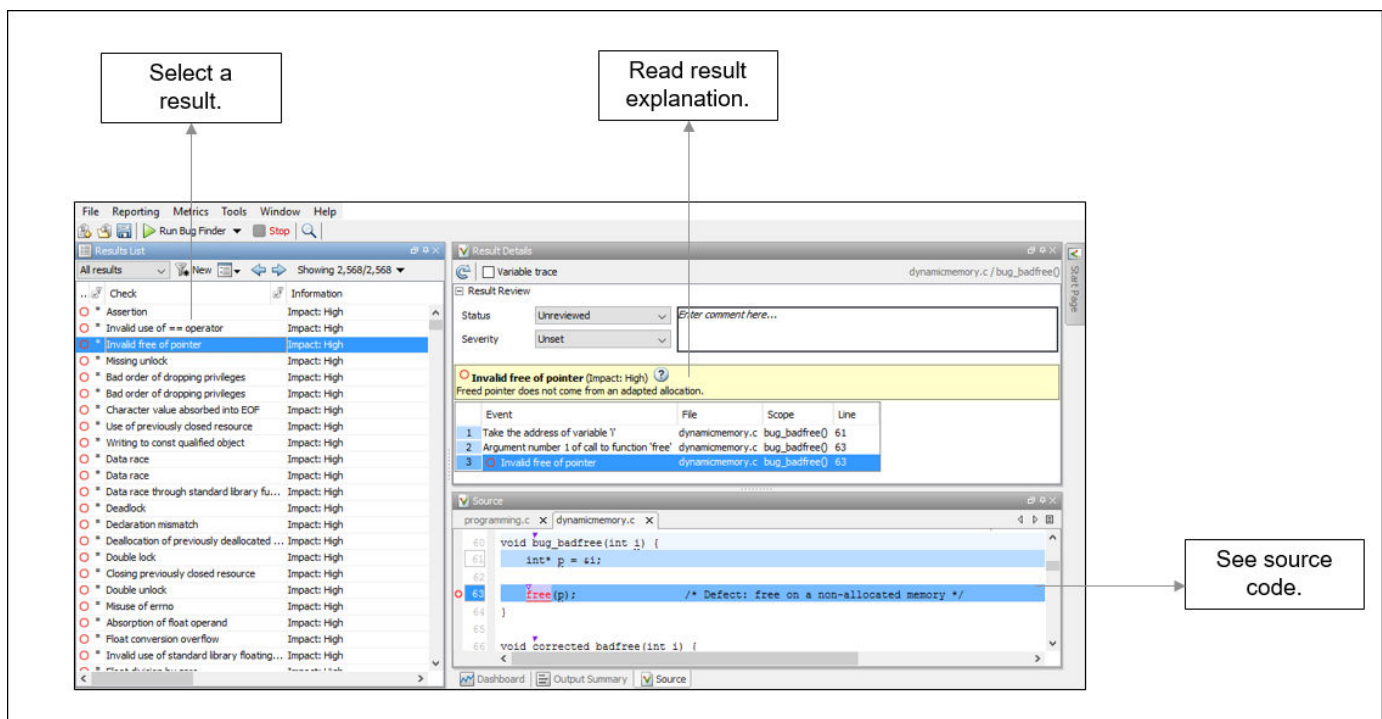
This topic shows how to review Bug Finder results in the user interface of the Polyspace desktop products. For a similar workflow in the Polyspace Access web interface, see “Interpret Bug Finder Results in Polyspace Access Web Interface” (Polyspace Bug Finder Access). To see how to review results of Polyspace as You Code in IDEs, see “Run Polyspace as You Code in IDEs and Review Results” (Polyspace Bug Finder Access).

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 details message. The interface is divided into several panes:

- Result Details:** Shows the message "Misuse of sign-extended character value (Impact: Medium)". It includes a description, severity, and a table of events.
- Source:** Shows the C++ code snippet where the bug occurred, including comments and line numbers.
- Contextual Help:** Provides a detailed description of the issue, a risk assessment, a fix, and examples. It also includes result information and external standards references.


Annotations with arrows point to specific parts of the interface:

- "Open contextual help." points to the help icon in the top right of the Result Details pane.
- "Read brief explanation." points to the message text in the Result Details pane.
- "Read detailed explanation with examples." points to the Description and Examples sections in the Contextual Help pane.
- "Check external standards." points to the See Also section in the Contextual Help pane.

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.

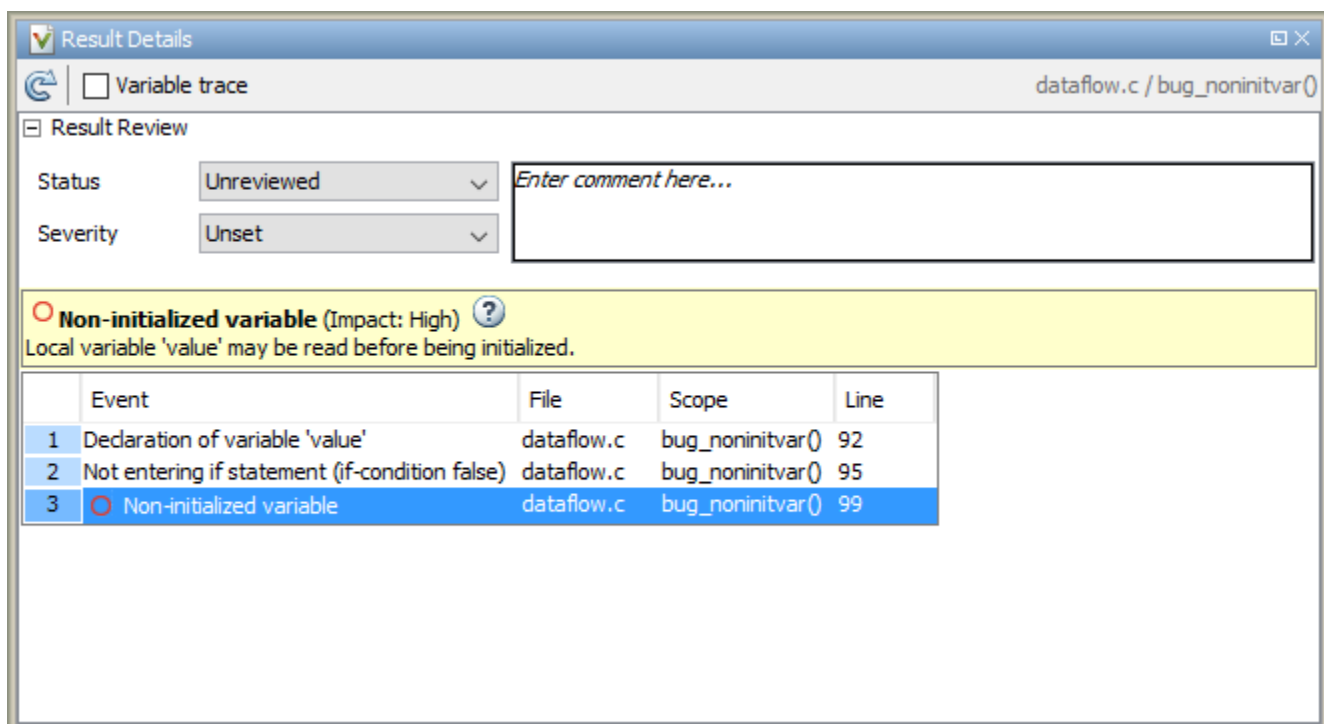
At this point, you might be ready to decide whether to fix the issue or not. Once you identify a fix, it might help to review all results of that type together.

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.



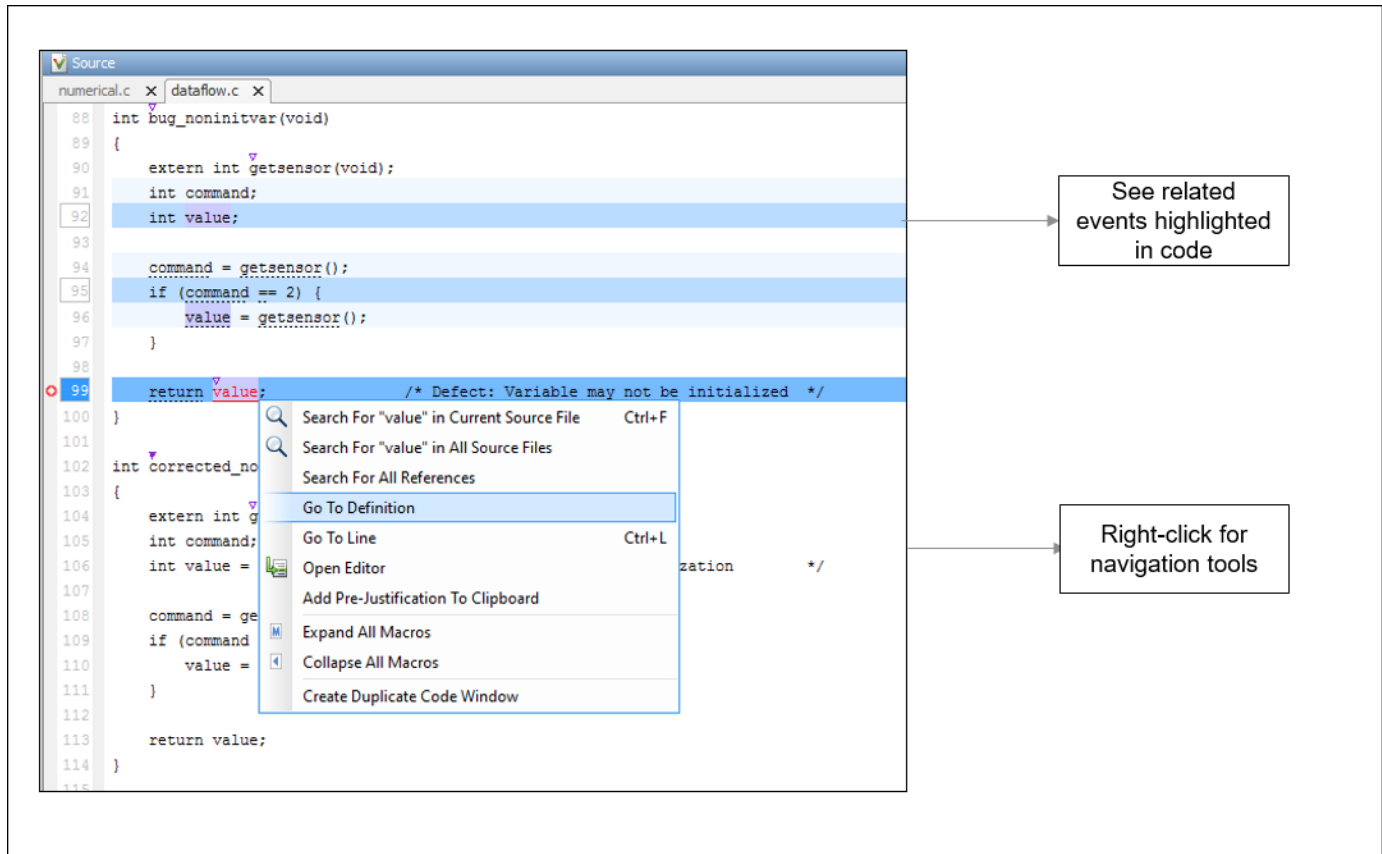
In the above event traceback, this sequence is shown:

- 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.

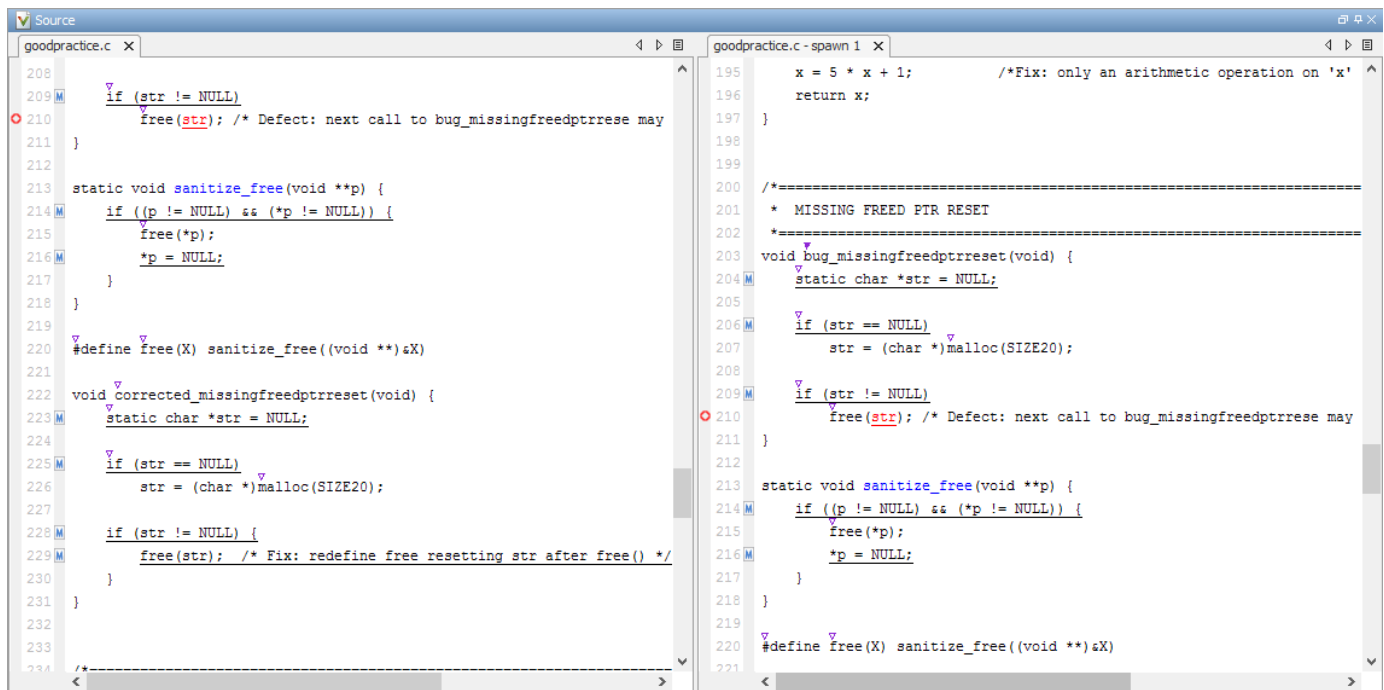
you might want to navigate from a function call to the function definition. Right-click the function and select **Go To Definition**.

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 Polyspace Results Through Bug Fixes or Justifications” on page 17-2
- “Filter and Group Results in Polyspace Desktop User Interface” on page 18-2

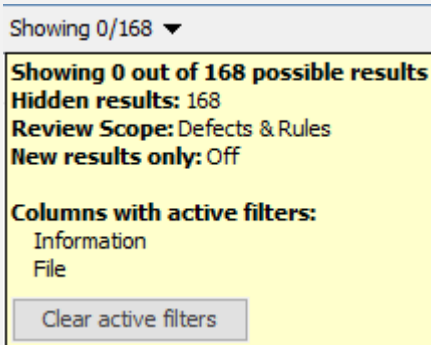
Investigate the Cause of Empty Results List

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” (Polyspace Bug Finder Access)*.

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 . 

See Also

More About

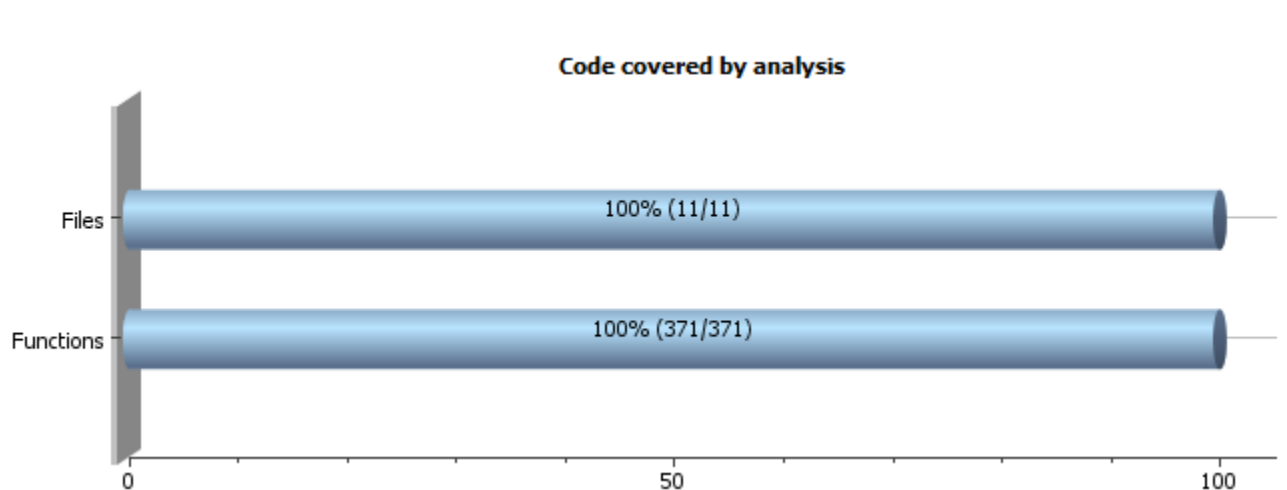
- “Address Polyspace Results Through Bug Fixes or Justifications” on page 17-2
- “Troubleshoot Compilation Errors”

Dashboard

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**



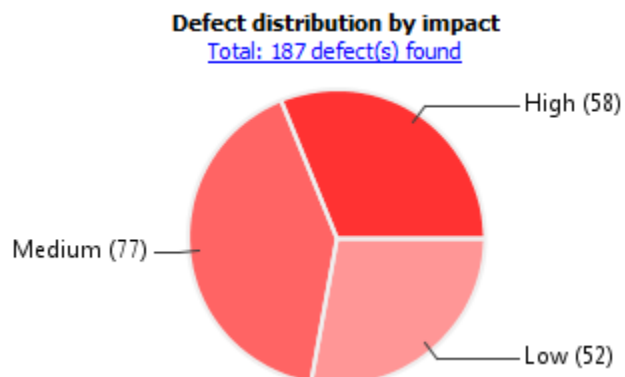
From this graph you can obtain the following information:

- **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.

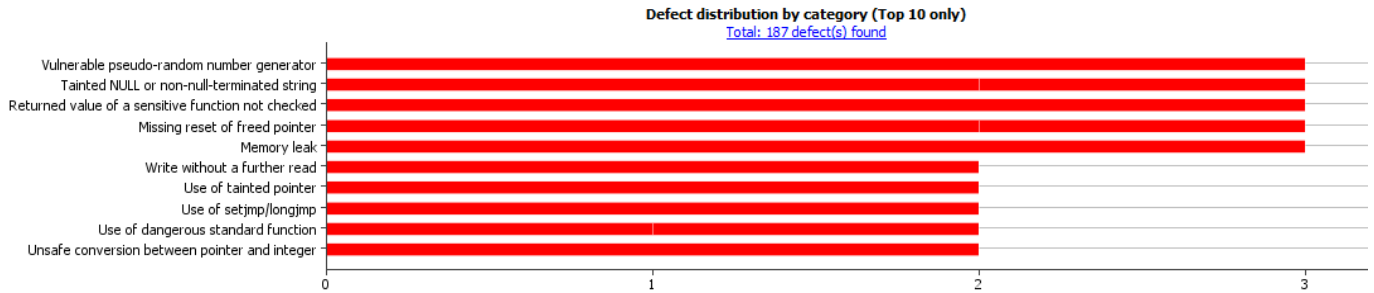
- **Functions:** Ratio of analyzed functions to total number of functions *in the analyzed files*. If the analysis of a function takes longer than a certain threshold value, Polyspace Bug Finder does not analyze the function.

- **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 18-9.

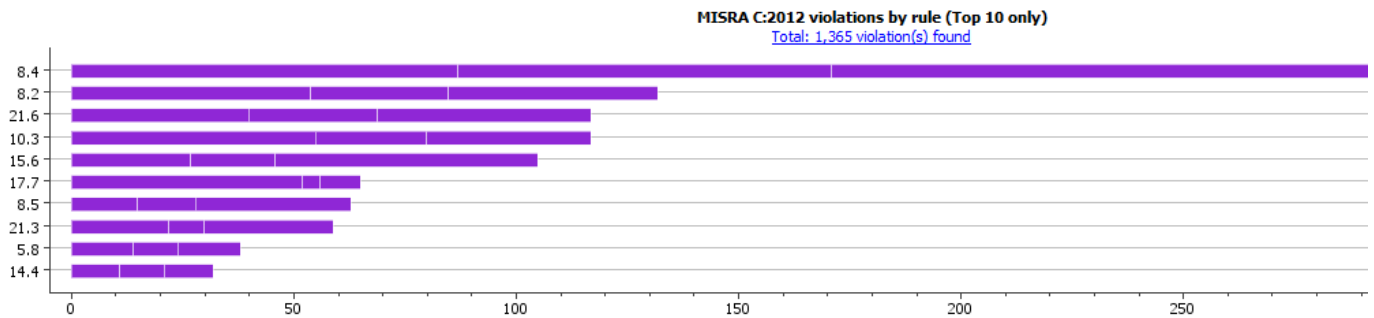
- **Defect distribution by category or file**



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, JSF, 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>

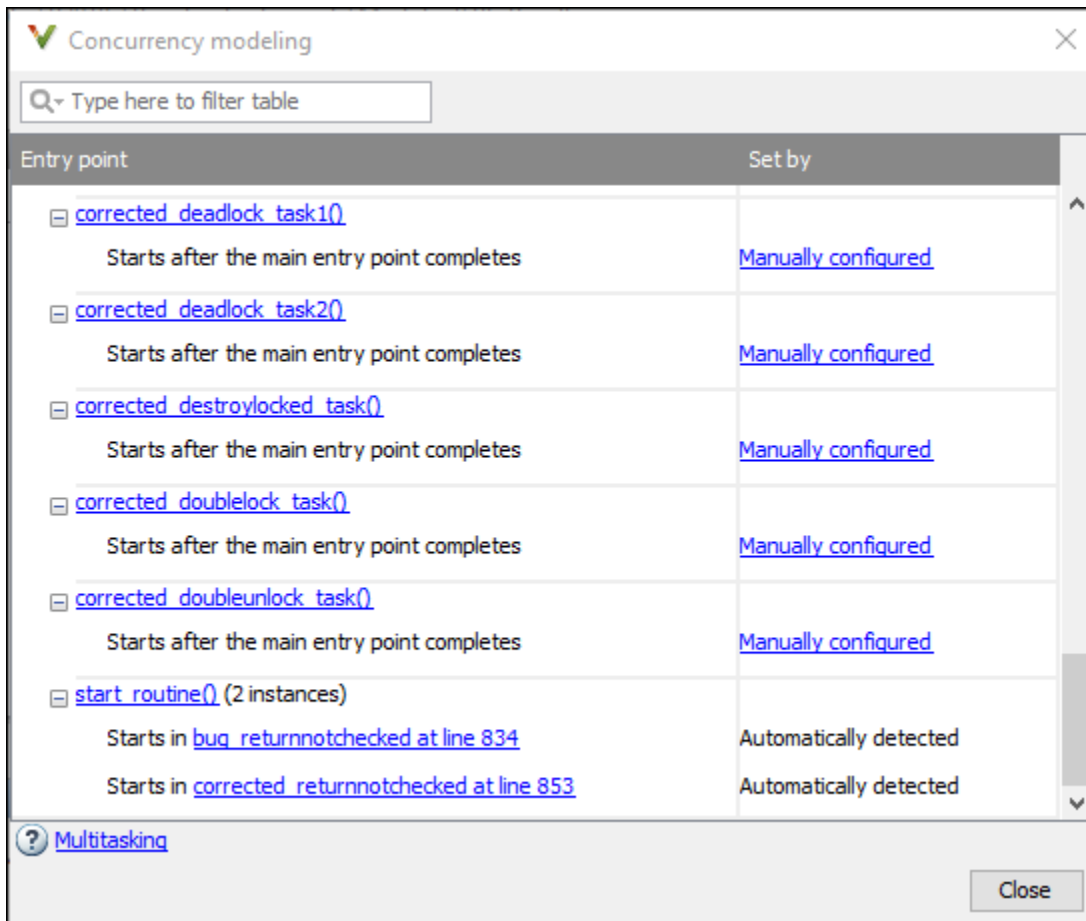
You can also perform the following actions on this pane:

- 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 18-2.

- 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 16-13**.

Concurrency Modeling

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 11-2
- “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 11-5
- “Configuring Polyspace Multitasking Analysis Manually” on page 11-16

Results List

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 14-40.
- **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. <p>For instance, MISRA C:2012 defines groups related to code constructs such as functions, pointers and arrays, etc.</p>
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. <p>For an example of how to use this column, see the result MISRA C:2012 Dir 1.1.</p>
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 .


Attribute	Description
Function	Function containing the instruction where the result occurs. If the function is a method of a class, it appears in the format <i>class_name::function_name</i> .
Folder	Path to the folder that contains the source file with the result
CWE ID	CWE IDs corresponding to the Bug Finder results. See “CWE Coding Standard and Polyspace Results” on page 14-78
Severity	Level of severity you have assigned to the result. The possible levels are: <ul style="list-style-type: none"> • Unset • High • Medium • Low
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
Comments	Comments you have entered about the result
Assigned to	User name of reviewer assigned to this result. This column is visible only for results that you open from Polyspace Access.
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. This column is visible only for results that you open from Polyspace Access.

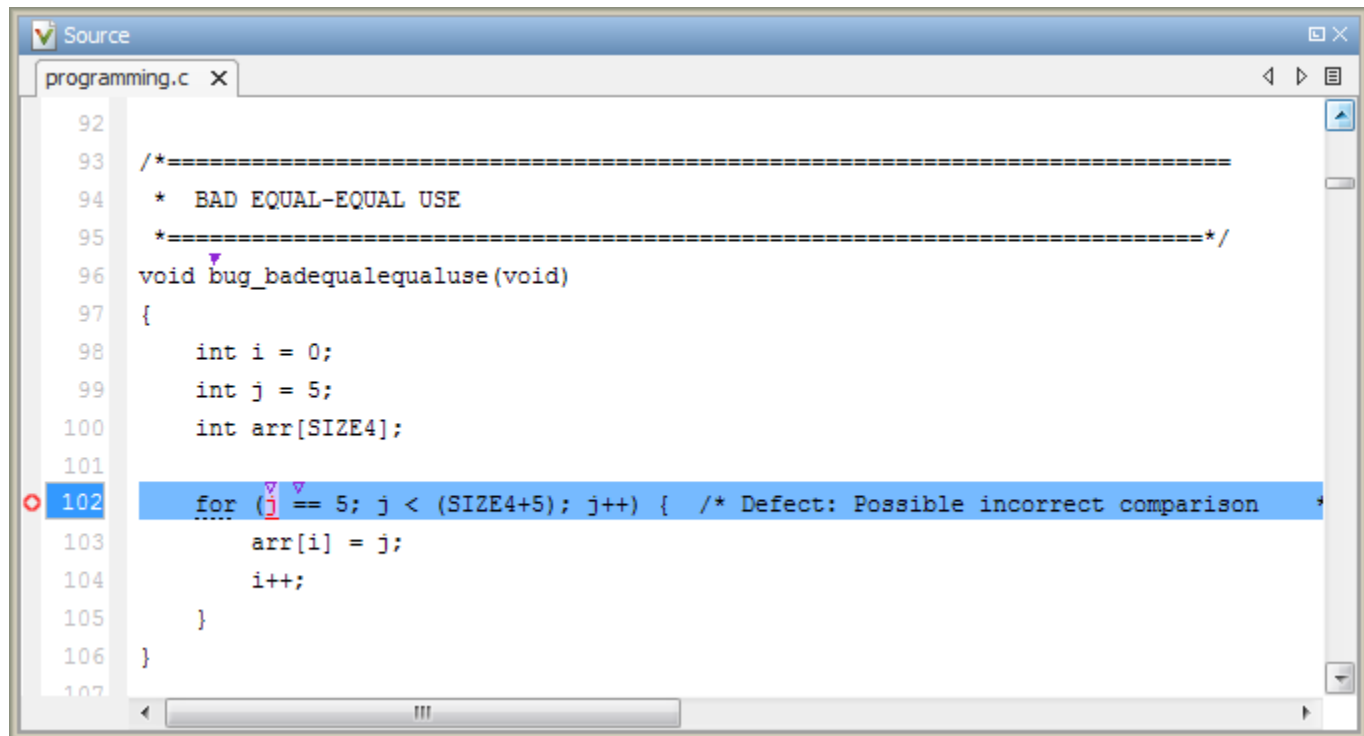
To show or hide any of the columns, right-click anywhere on the column titles. From the context menu, select or clear the title of the column that you want to show or hide.

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 18-2.

Source

The **Source** pane shows the source code with the defects colored in red and the corresponding line number marked by .



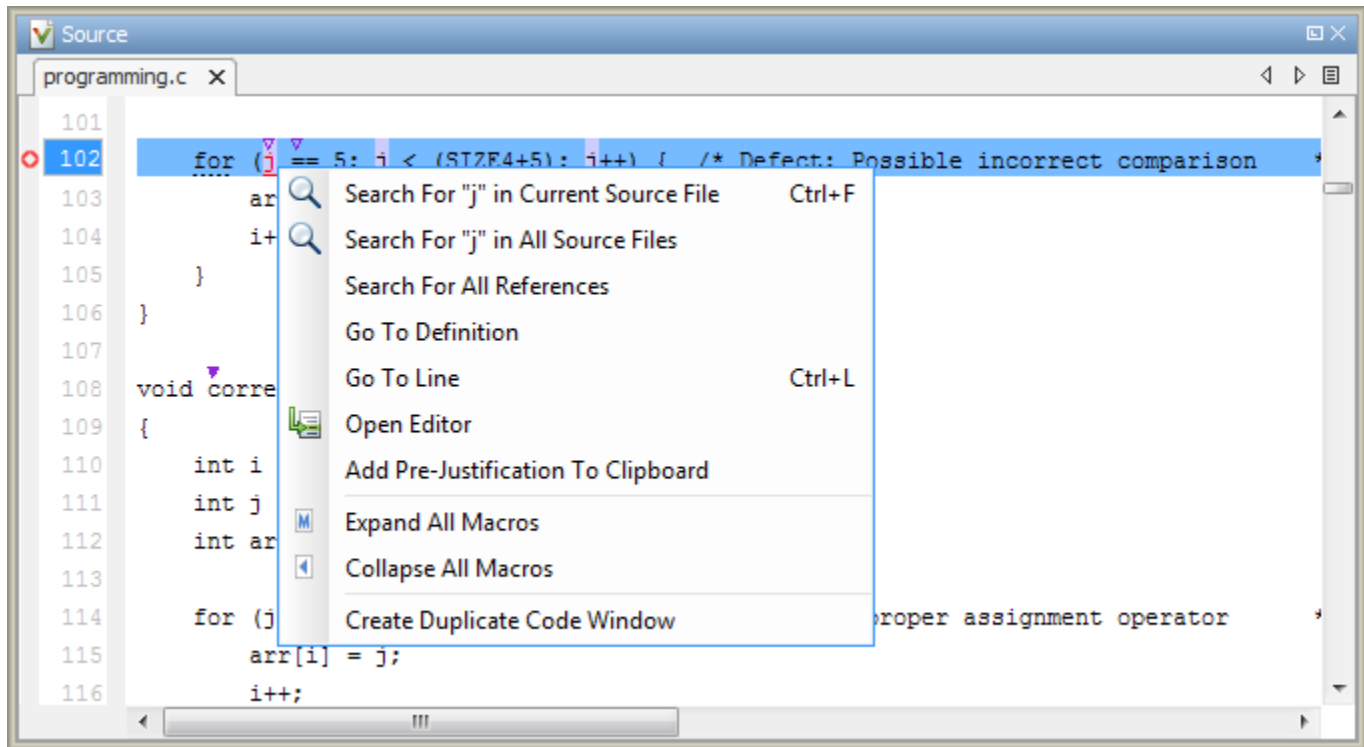
```
92
93 /*=====
94  * BAD EQUAL-EQUAL USE
95  *=====*/
96 void bug_badequalequaluse(void)
97 {
98     int i = 0;
99     int j = 5;
100    int arr[SIZE4];
101
102    for (j) == 5; j < (SIZE4+5); j++) { /* Defect: Possible incorrect comparison */
103        arr[i] = j;
104        i++;
105    }
106 }
107
```

Tooltips

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:

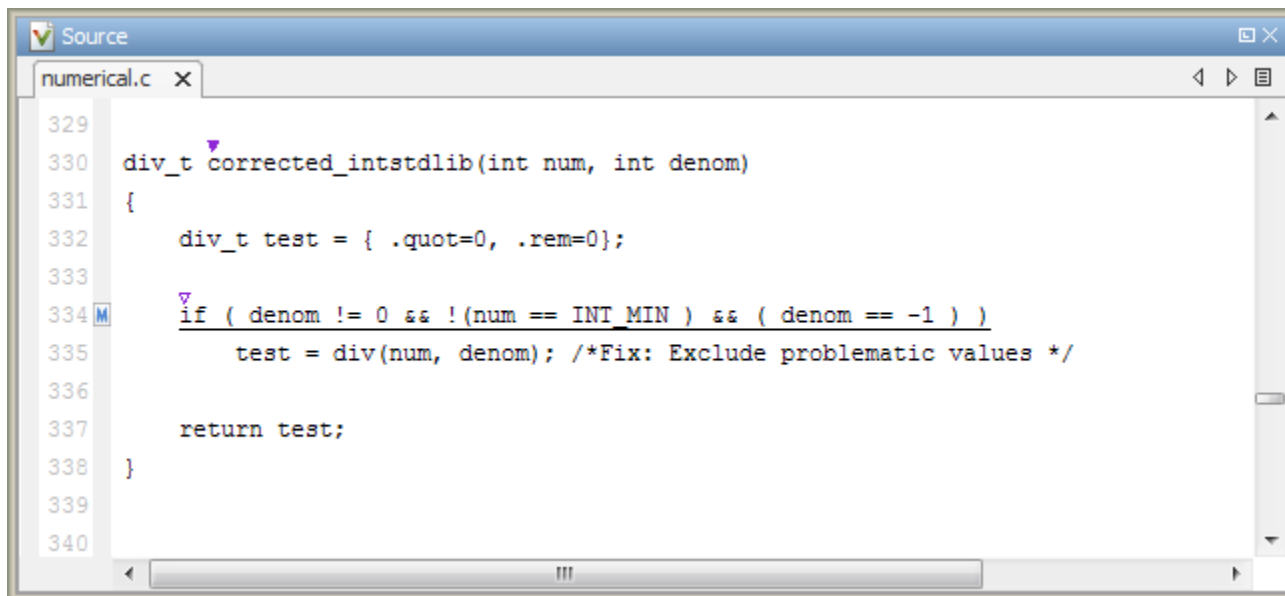


For example, if you right-click the variable `i`, you can use the following options to examine and navigate through your code:

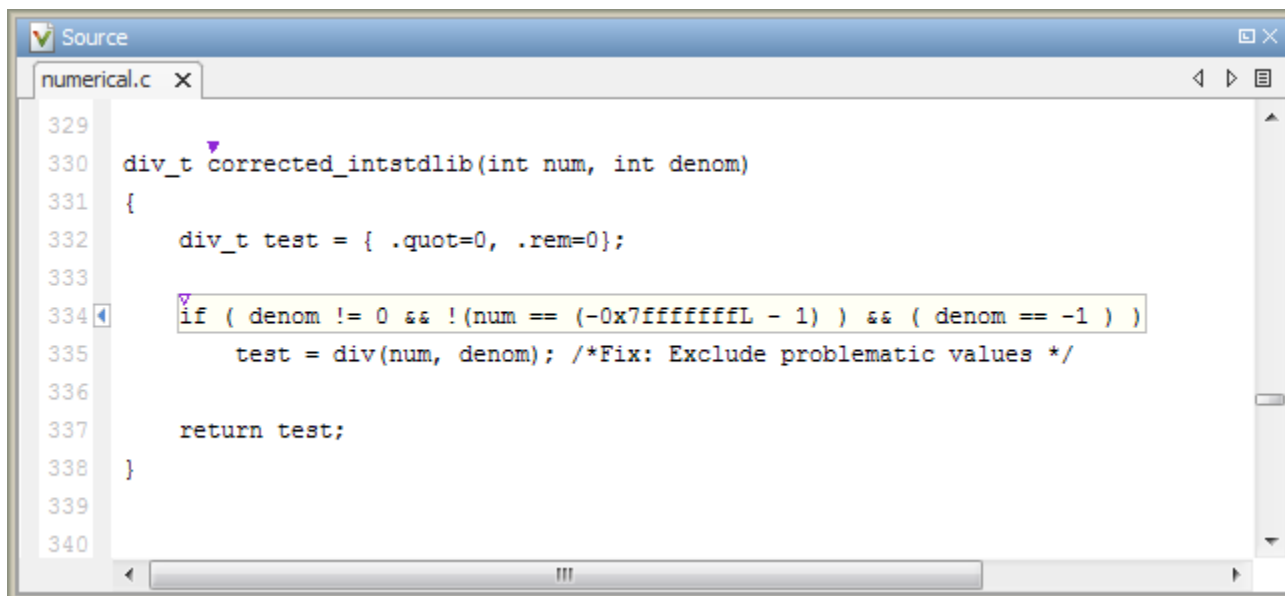
- **Search "j" in Current Source** — List occurrences of the string within the current source file on the **Search** pane.
- **Search "j" in All Source Files** — List occurrences of the string within the source files on the **Search** pane.
- **Search For All References** — List all references in the **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.



When you click a line with this icon, the software displays the contents of macros on that line in a box.



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**.

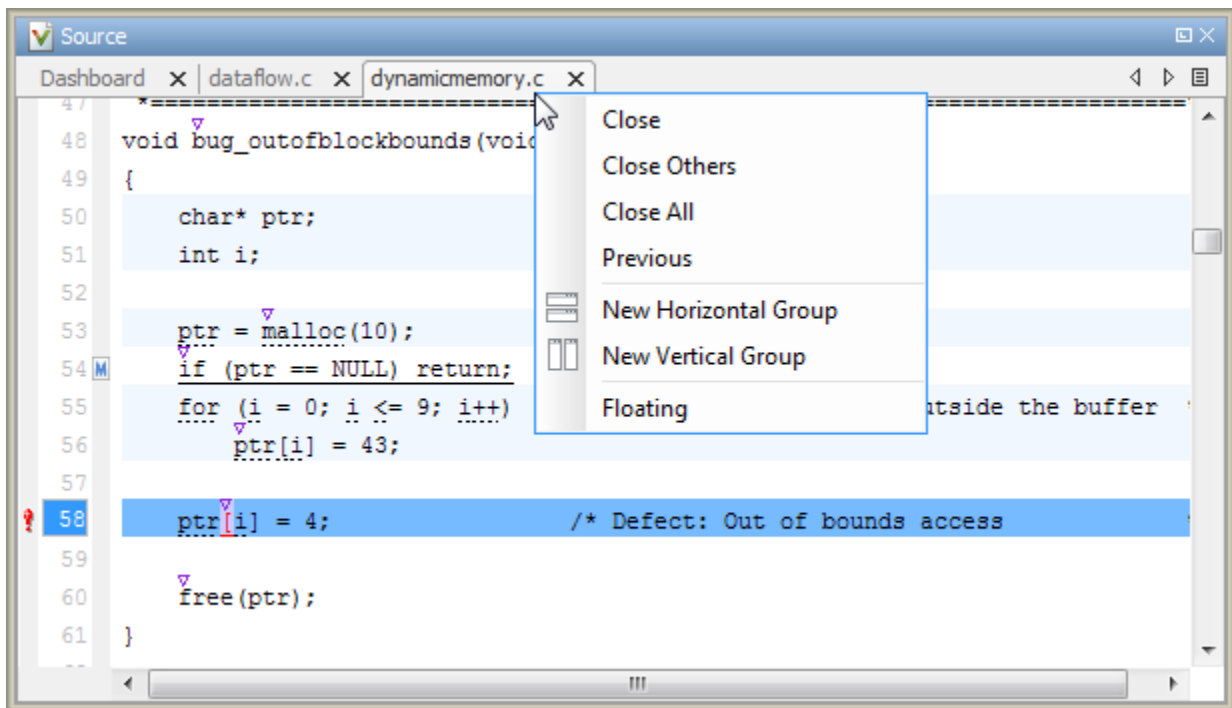
Note

- 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.

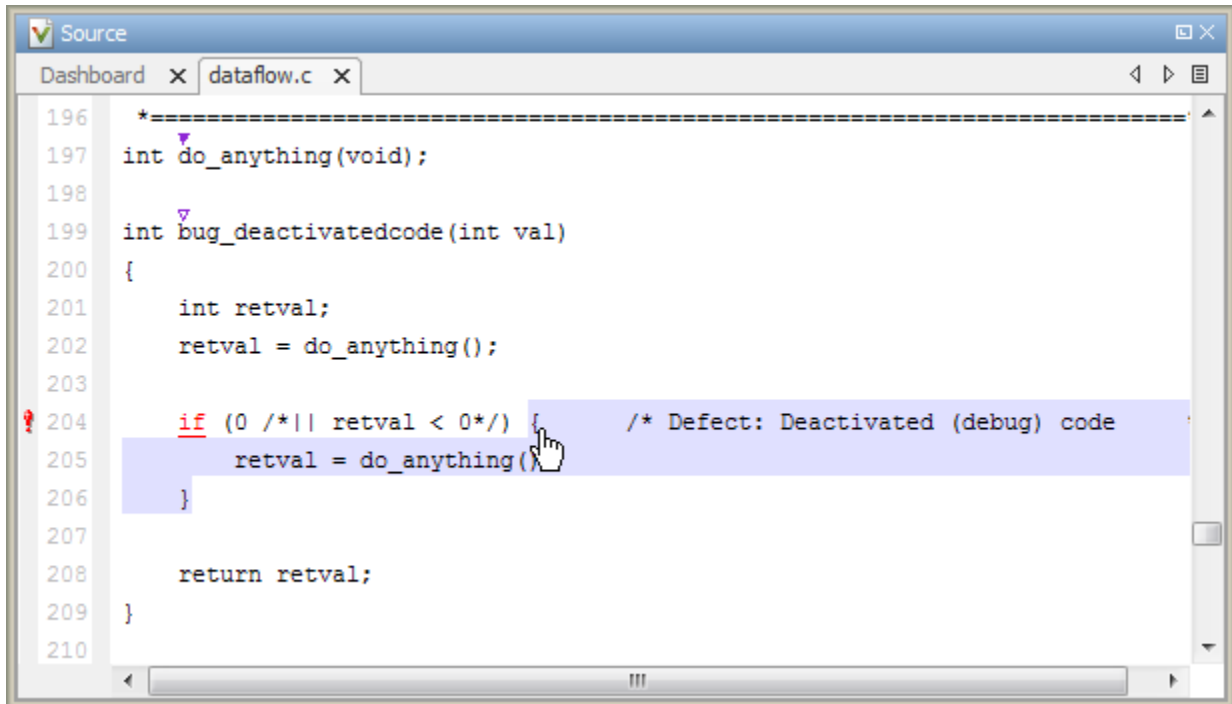


From the **Source** pane context menu, you can:

- **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.




The screenshot shows a window titled "Source" with a tab for "dataflow.c". The code is as follows:

```
196  *=====
197  int do_anything(void);
198
199  int bug_deactivatedcode(int val)
200  {
201      int retval;
202      retval = do_anything();
203
204      if (0 /*|| retval < 0*/) { /* Defect: Deactivated (debug) code
205          retval = do_anything();
206      }
207
208      return retval;
209  }
210
```

The code block from line 204 to 206 is highlighted in blue. A mouse cursor is pointing at the closing brace of the `if` statement on line 206.


Call Hierarchy

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.

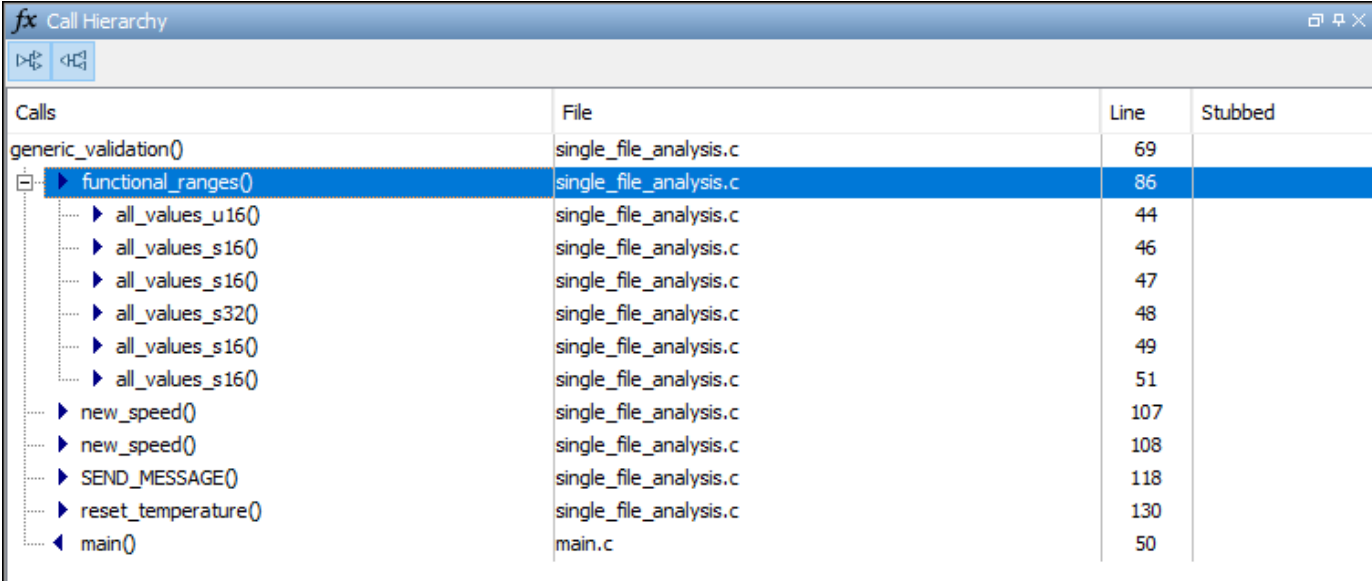
Note In Polyspace Bug Finder, you might not see all callers or callees of a function, especially for calls through function pointers and dead code.

For instance, Polyspace Bug Finder 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 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

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**

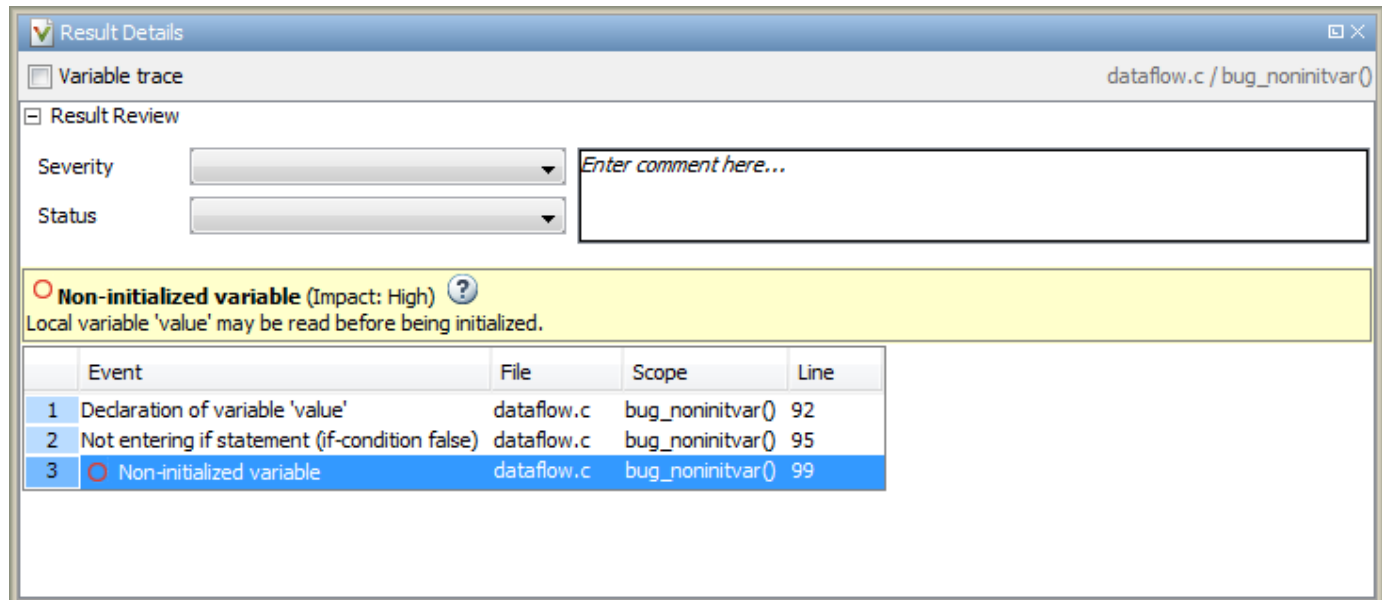
From the **Stubbed** column, you can determine if a function is stubbed. The entries in the column show why a function was stubbed.


- **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`.

Result Details

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” (Polyspace Bug Finder Access).

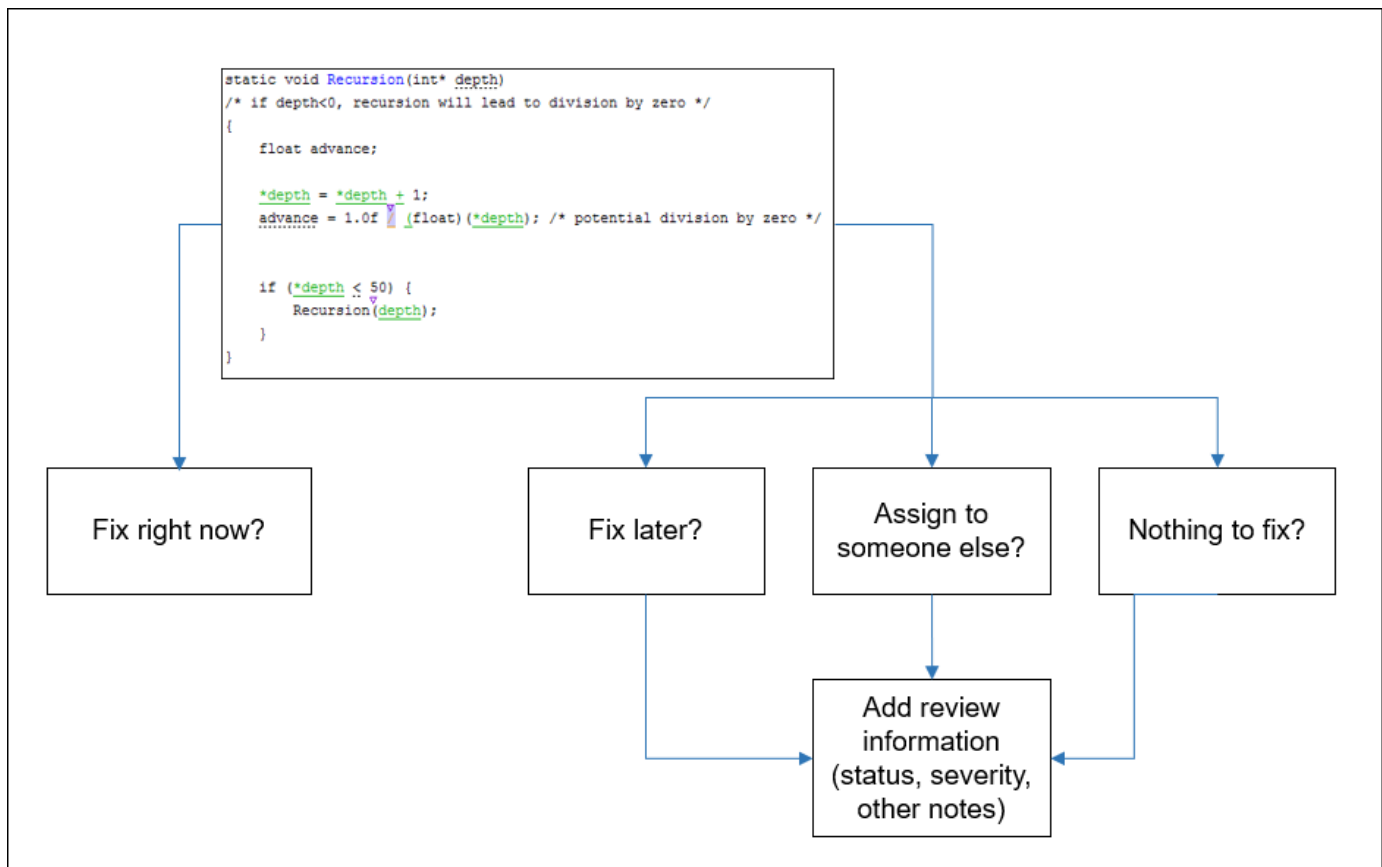
Fix or Comment Polyspace Results

- “Address Polyspace Results Through Bug Fixes or Justifications” on page 17-2
- “Annotate Code and Hide Known or Acceptable Results” on page 17-6
- “Short Names of Code Complexity Metrics” on page 17-13
- “Annotate Code for Known or Acceptable Results (Not Recommended)” on page 17-15
- “Define Custom Annotation Format” on page 17-19
- “Annotation Description Full XML Template” on page 17-27

Address Polyspace Results 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” (Polyspace Bug Finder Access).

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

The screenshot shows the 'Result Details' pane in Polyspace. It includes a 'Variable trace' checkbox, a 'Result Review' section with 'Severity' set to 'High' and 'Status' set to 'Fix', and a text area containing the comment 'Adding missing else condition.'. Below this is a yellow warning banner for a 'Non-initialized pointer' error (Impact: High) with the message 'Local pointer 'pi' may be read before being initialized.'. At the bottom is a table of events:

	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

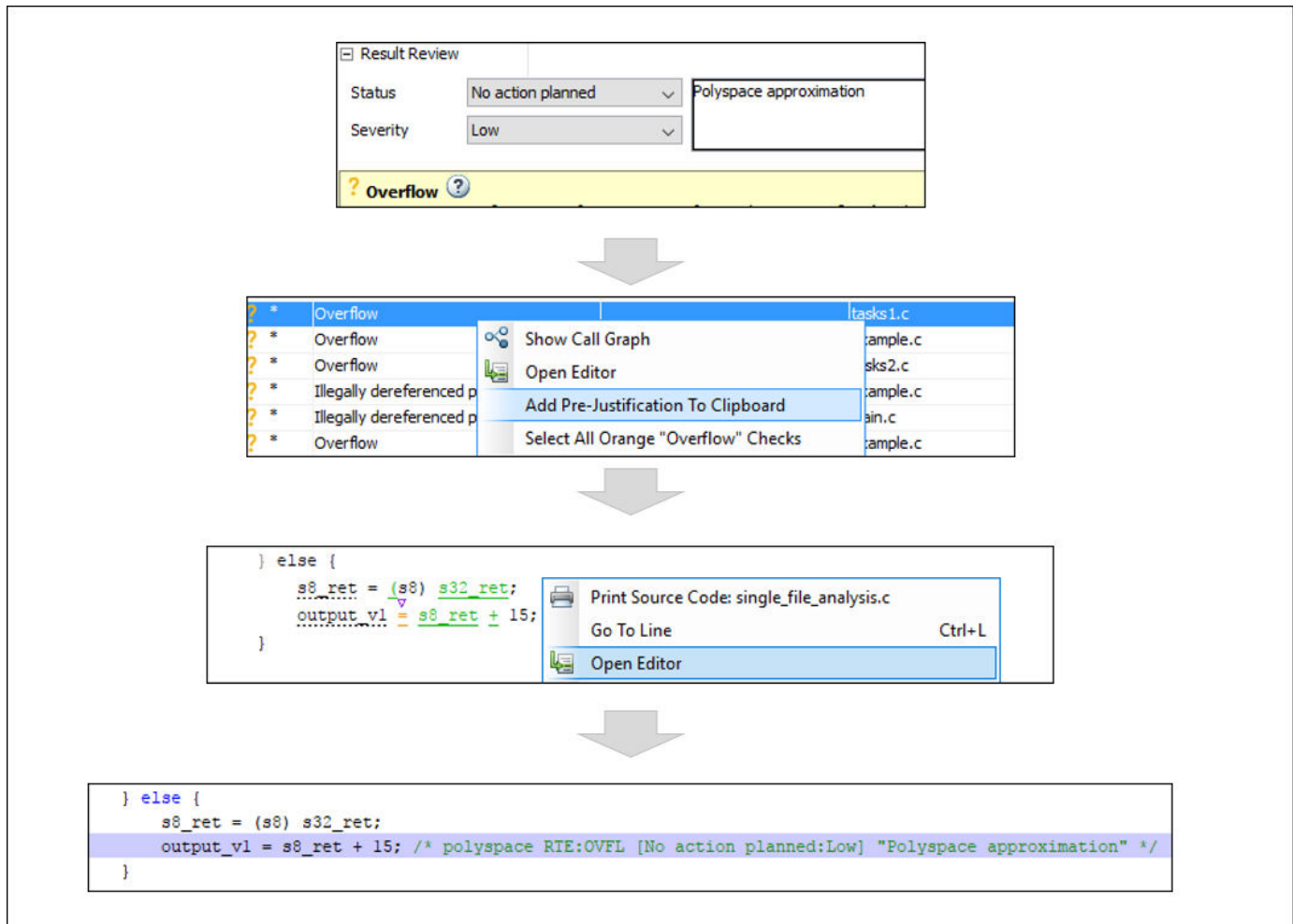
You can add the information either on the **Results List** or **Result Details** pane. Select a result, then set the **Severity** and **Status** fields, and optionally, enter notes with more explanations. The status indicates your response to the Polyspace result. If you do not plan to fix your code in response to a result, assign one of the following statuses:

- Justified
- No Action Planned
- Not a Defect

You can create your own statuses to assign. To create a new status, go to **Tools > Preferences** and select the **Review Statuses** tab.

Based on the status, Polyspace considers that you have given due consideration and justified that result (retained the code despite the result).

Comment or Annotate in Code



If you enter code comments or annotations in a specific syntax, the software can read them and populate the **Severity**, **Status**, and **Comment** fields in the next analysis of the code.

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. If you follow this workflow, Polyspace assumes that you have set a status of **No Action Planned**. The software hides the result from all places (except reports needed for certification). The only exceptions are the safety-critical Code Prover run-time checks, which are hidden from the results list but not the source code.

To unhide the hidden results, from the **Showing** menu, clear the box **Hide results justified in code**.

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

If you want to explicitly set a status, first fill the **Status** field for a result and then copy to your code. Paste on the line containing the result.

If you want to directly type the annotation, see the annotation syntax in “Annotate Code and Hide Known or Acceptable Results” on page 17-6.

See Also

More About

- “Annotate Code and Hide Known or Acceptable Results” on page 17-6
- “Import Review Information from Previous Polyspace Analysis” on page 15-2

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 the Polyspace user interface or by typing them directly in your code. For the general workflow, see “Address Polyspace Results Through Bug Fixes or Justifications” on page 17-2. 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 */
```

Annotations begin with the keyword `polyspace` and must include `Family` and `Result_name` field values. You can optionally specify `Status`, `Severity`, and `Comment` field values.

```
polyspace Family:Result_name [Status:Severity] "Comment"
```

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 17-7 and “Syntax Examples” on page 17-10.

Annotating Code Block

To annotate a block of code, use the following syntax.

- Annotation for current line of code and `n` following lines:

```
code; /* polyspace +n Family:Result_name */
```

- Annotation for block of code:

```
/* polyspace-begin Family:Result_name */  
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] "Block annotation"*/
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] "Nested annotation applied"*/
10        return x;
11    return x;
12 }
13 /*polyspace-end MISRA-C:14.9 [To fix:High] "Block annotation"*/

```

For further details, see “Annotation Syntax Details” on page 17-7 and “Syntax Examples” on page 17-10.

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 17-7 and “Syntax Examples” on page 17-10.

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 17-10.

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 • 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 • CUSTOM for violations of custom coding rules <p>To specify all analysis results, use the asterisk character * : *.</p> <p>See “Syntax Examples” on page 17-17.</p>

Field	Allowed Value
<i>Result_name</i>	<p>For DEFECT, use short names of checkers. See “Short Names of Bug Finder Defect Checkers” on page 14-26.</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 17-13.</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>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 17-17.</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.

```
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. code ; // polyspace +3 MISRA2012:*
5. //comment
6.
7. code;
8. 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.

```
some 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 */
```

Add Explanatory Comments 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
code; /* polyspace DEFECT:BAD_FREE MISRA2004:* "OK Defect and MISRA" */
//Multiple comments incorrect syntax:
code; /* polyspace DEFECT:* "OK defect" MISRA2004:5.2 "OK MISRA" */
//Multiple comments correct syntax:
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 17-6 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
code; /* polyspace DEFECT:* [To fix] "some comment" */

//Set Status 'To fix' and Severity 'High'
code; /* polyspace VARIABLE:* [To fix: High] "some comment"*/

//Set custom status 'Assigned' and Severity 'Medium'
code; /* polyspace MISRA2012:12.* [Assigned: Medium] */
```

See Also

-xml-annotations-description

More About

- “Define Custom Annotation Format” on page 17-19
- “Short Names of Bug Finder Defect Checkers” on page 14-26
- “Short Names of Code Complexity Metrics” on page 17-13

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 Protected Shared Variables (Code Prover only)	PSHV
Number of Recursions	AP_CG_CYCLE
Number of Potentially Unprotected Shared Variables (Code Prover only)	UNPSHV
Program Maximum Stack Usage (Code Prover only)	PROG_MAX_STACK
Program Minimum Stack Usage (Code Prover only)	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 Local Variable Size	LOCAL_VARS_MAX
Language Scope	VOCF
Lower Estimate of Local Variable Size	LOCAL_VARS_MIN
Minimum Stack Usage (Code Prover only)	MIN_STACK
Maximum Stack Usage (Code Prover only)	MAX_STACK
Number of Call Levels	LEVEL

Code Metric	Acronym
Number of Call Occurrences	NCALLS
Number of Called Functions	CALLS
Number of Calling Functions	CALLING
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 17-6

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 17-6.

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 18-2.

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 17-17.

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 Checkers” on page 14-26 • “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 Metrics 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 17-25.

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 17-6.	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 17-6.	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 17-27
- “Annotate Code and Hide Known or Acceptable Results” on page 17-6

- “Resolve -xml-annotations-description Errors” on page 21-54

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 17-19.

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 17-25.
	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 17-6.
	Result_Name	Required	Corresponds to Polyspace result names. For a list of allowed values, see allowed values on page 17-6.

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 17-6

Manage Results

- “Filter and Group Results in Polyspace Desktop User Interface” on page 18-2
- “Classification of Defects by Impact” on page 18-9

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” (Polyspace Bug Finder Access).

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.

The diagram illustrates the Polyspace Desktop user interface for managing results. It shows three windows of the 'Results List' interface.

The top window shows the full list of results, with columns for Family, Group, Check, File, and Function. The status bar indicates 'Showing 397/397'.

Two arrows point from the top window to the bottom windows:

- Filter results:** The bottom-left window shows a filtered list of results, with the status bar indicating 'Showing 5/397'. The results are:
 - 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:** The bottom-right window shows results grouped by file. The file 'example.c' is expanded, showing functions:
 - 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.

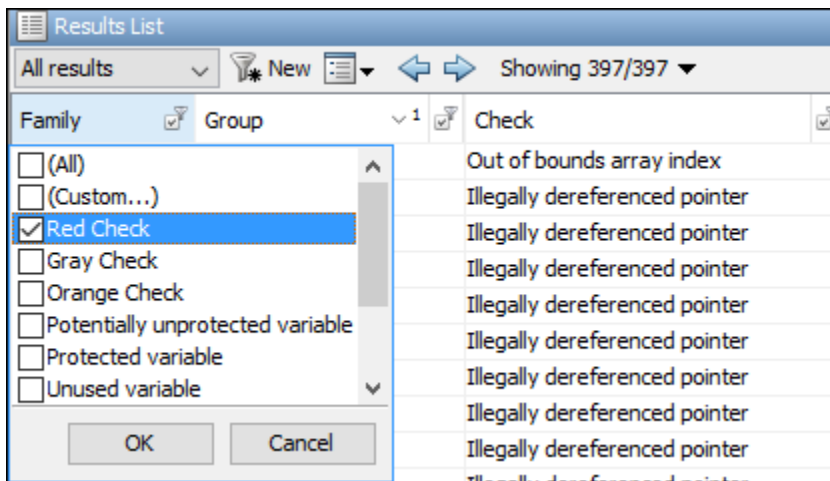
For instance, in Bug Finder, you can display only high-impact defects. See “Classification of Defects by Impact” on page 18-9.

- You can display only new results found since the last analysis.
- You can display only the results that have not justified.

For information on justification, see “Address Polyspace Results Through Bug Fixes or Justifications” on page 17-2.

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 information on the columns, see:

- “Results List” on page 16-15
- “Results List” (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 15-2.

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

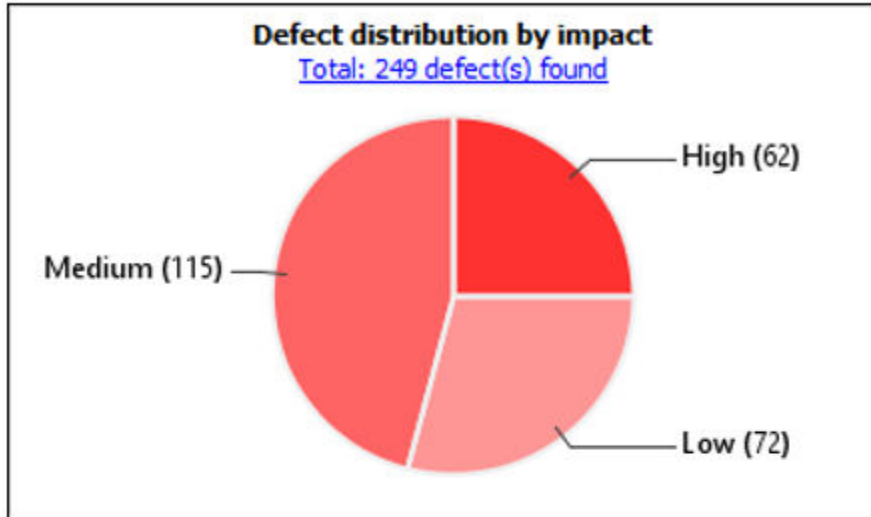
the **Folder** column, select **Custom** from the filter dropdown. Specify the root folder name for the doesn't contain filter.

You can use wildcard characters for the custom filter. The wildcard ? represents 0 or 1 character and * represents 0 or more characters.

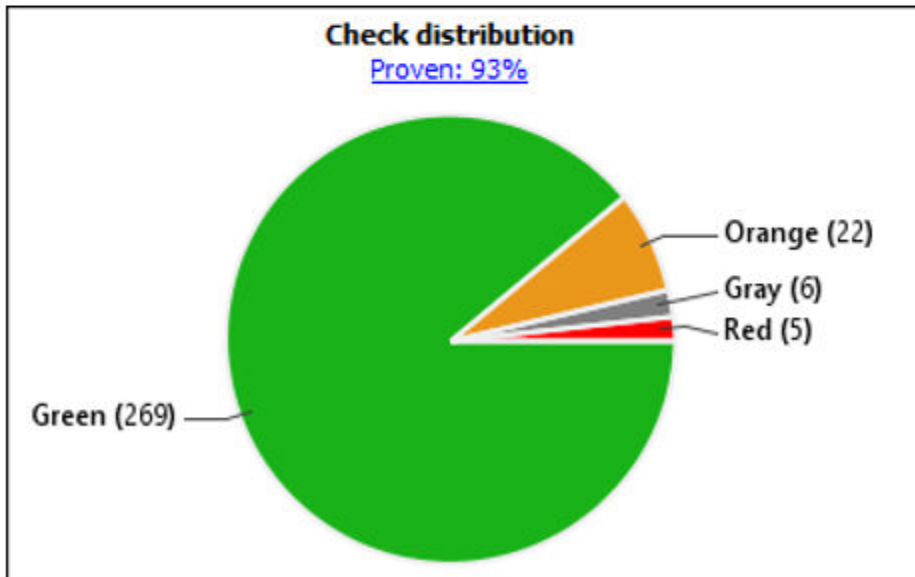
If you apply filters in this way, they carry over to the next analysis. You can also name and save a subset of filters for use in multiple projects. To apply the named set of filters, pick this filter set from the **All results** list. To create a new entry in this list, select **Tools > Preferences** and create your own set of filters on the **Review Scope** tab.

Filter Using Dashboard

Bug Finder



Code Prover



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.

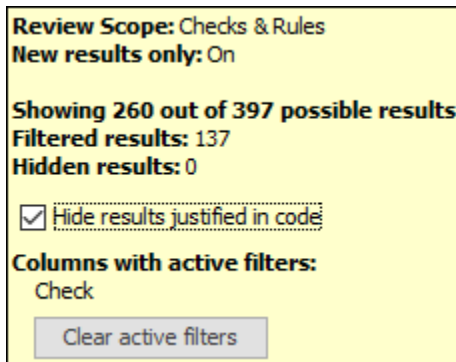
For instance, in this code, the unknown value `input` can cause an overflow and a division by zero. The variable `input` is an orange source that causes two orange checks.

```
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.

Orange Sources				
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_j	example.c	27	2
local volatile variable	all_values_s32.tmps32	single_file_analysis.c	29	2

See Filters Used



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:

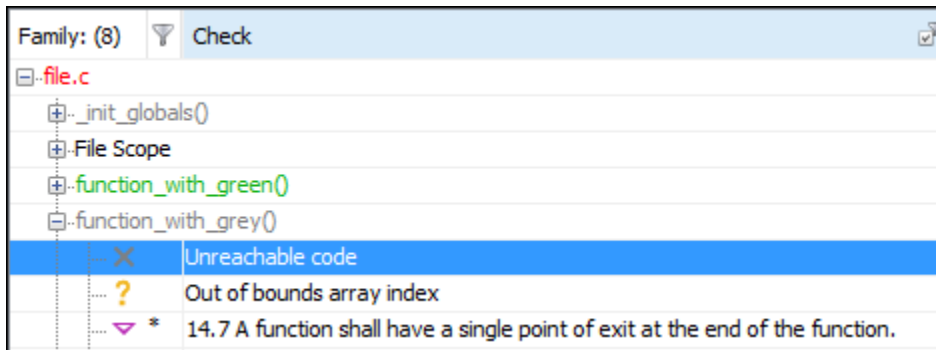
- **Review Scope:** If you pick a named set of filters from the **All results** dropdown, you see this filter set.
- **New results only:** If you use the **New** button to see only new results, you see this filter enabled.
- **Filtered results:** You see the number of results filtered in the Polyspace 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 Polyspace Results Through Bug Fixes or Justifications” on page 17-2.

- **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 18-9

Classification of Defects by Impact

To prioritize your review of Polyspace Bug Finder defects, you can use the **Impact** attribute assigned to the defect. This attribute appears on:

- The **Dashboard** pane, in a **Defect distribution by impact** pie chart.

You can view at a glance whether you have many high impact defects. You can also select elements on the chart to filter out low or medium impact defects from the **Results List** pane. See “Filter and Group Results in Polyspace Desktop User Interface” on page 18-2.

- The **Results List** pane, in the **Information** column. When you select **None** from the  list, the defects are sorted by impact.

You can filter out low and/or medium impact defects using this column or through the **Review Scope** tab in your preferences. See “Filter and Group Results in Polyspace Desktop User Interface” on page 18-2.

- The **Result Details** pane, beside the defect name.

The impact 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

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

Programming

- Assertion
- Character value absorbed into EOF
- Declaration mismatch
- Errno not reset
- Incorrect value forwarding
- Invalid use of == (equality) 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
- 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
- Privilege drop not verified
- Returned value of a sensitive function not checked
- 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
- Data race including atomic operations
- 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 conditional 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
- Byte-wise 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 `c_str()` to `std::string` construction
- Expensive constant `std::string` construction
- Expensive copy in a range-based for loop iteration
- Expensive local variable copy
- Expensive logical operation
- Expensive pass by value
- Expensive return by value
- Inefficient string length computation
- Missing `constexpr` specifier
- `std::endl` may cause an unnecessary flush
- `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
- 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 = (assignment) 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
- 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
- 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 a same data
- C++ reference to const-qualified type with subsequent modification
- C++ reference type qualified with const or volatile
- Delete of void pointer
- 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
- Line with more than one statement
- Macro terminated with a semicolon
- Macro with multiple statements
- Missing break of switch case
- Missing overload of allocation or deallocation function
- Missing reset of a freed pointer
- Possibly inappropriate data type for switch expression
- Redundant expression in sizeof operand
- Semicolon on same line as if, for or while statement
- Unmodified variable not const-qualified
- Unused parameter
- Use of a forbidden function
- Use of setjmp/longjmp

Numerical

- Float overflow
- Integer precision exceeded
- Possible invalid operation on boolean operand
- Precision loss from 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 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
- `std::endl` may cause an unnecessary flush
- 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
- 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
- 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 string to numeric value conversion

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 18-2

Generate Reports from Polyspace Results

- “Generate Reports” on page 19-2
- “Export Polyspace Analysis Results” on page 19-5
- “Export Polyspace Analysis Results to Excel by Using MATLAB Scripts” on page 19-7
- “Visualize Bug Finder Analysis Results in MATLAB” on page 19-9
- “Customize Existing Bug Finder Report Template” on page 19-13

Generate Reports

This example shows how to generate reports from Polyspace analysis results.

To generate reports, 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”.
- Export your results to a text file and generate graphs and statistics. See “Export Polyspace Analysis Results” on page 19-5.

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.

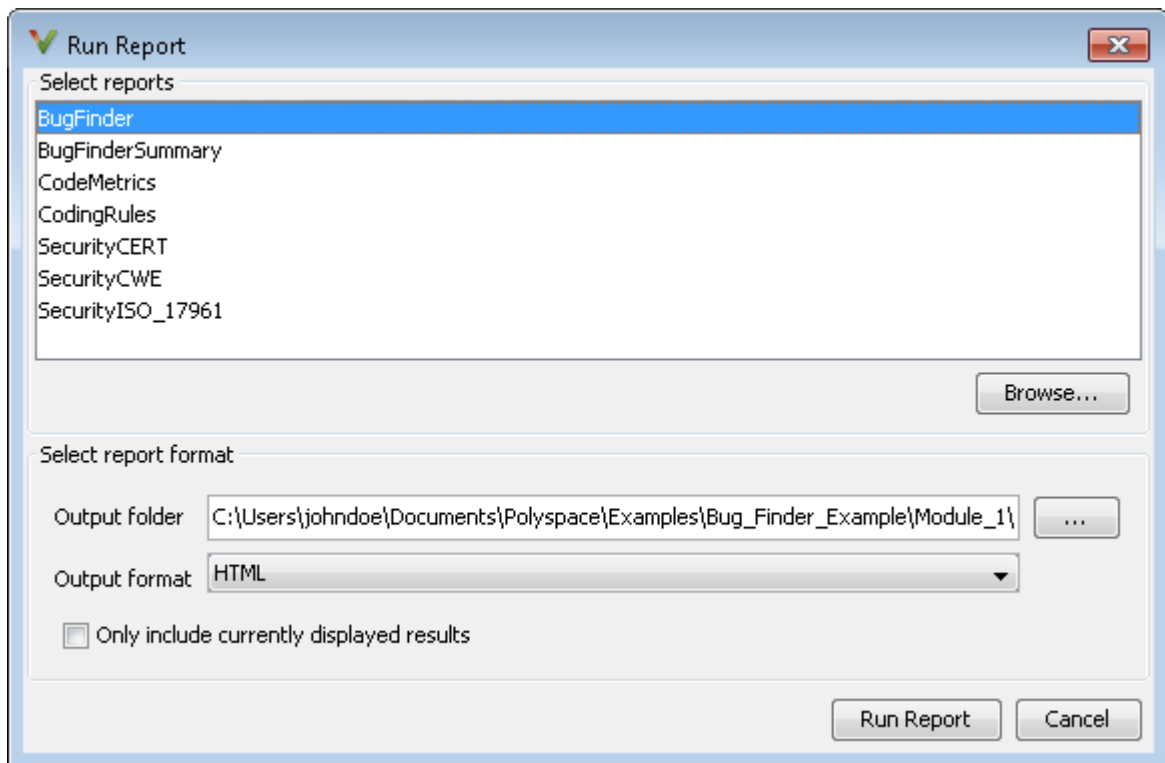
In Bug Finder, the report does not contain the line or column number for a result. Use the report for archiving, gathering statistics and checking whether results have been reviewed and addressed (for certification purposes or otherwise). To review a result in your source code, use the Polyspace user 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.



- 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 18-2.

- 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 19-13
- “Export Polyspace Analysis Results” on page 19-5

Export Polyspace Analysis Results

You can export your analysis results to a tab delimited 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 from the user interface or command line.

User Interface	Command Line
<ol style="list-style-type: none"> 1 Open your analysis results. 2 Export all results or only a subset of the results. <ul style="list-style-type: none"> • 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. <p>For more information on filtering, see “Filter and Group Results in Polyspace Desktop User Interface” on page 18-2.</p> 3 Select a location to save the text file and click OK. 	<p>Use the option <code>-format csv</code> with the <code>polyspace-results-export</code> command. For more information, see <code>polyspace-results-export</code>.</p>

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 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 19-9
- “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 text file or the table contains the result information available on the **Results List** pane in the user interface (except for line and column information). See:

- “Results List” on page 16-15
- “Results List” (Polyspace Code Prover)

Some differences in presentation between the **Results List** pane and the text file are listed below.

- The text file has a **New** column that shows whether the result is new compared to the last analysis on the same code.
- The text file or the table also contains a **Key** column. The entry in this column is based on the result name and the location of the result in a file.

When you merge analysis results from multiple modules that contain common files, use this entry 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 containing files and keys of results. If two results have the same files and keys, one is a duplicate of the other.

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.

See Also

`polyspace-results-export`

Related Examples

- “Visualize Bug Finder Analysis Results in MATLAB” on page 19-9
- “Export Polyspace Analysis Results to Excel by Using MATLAB Scripts” on page 19-7

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 19-5. 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 19-5

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 3-9.

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 14-40, 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:

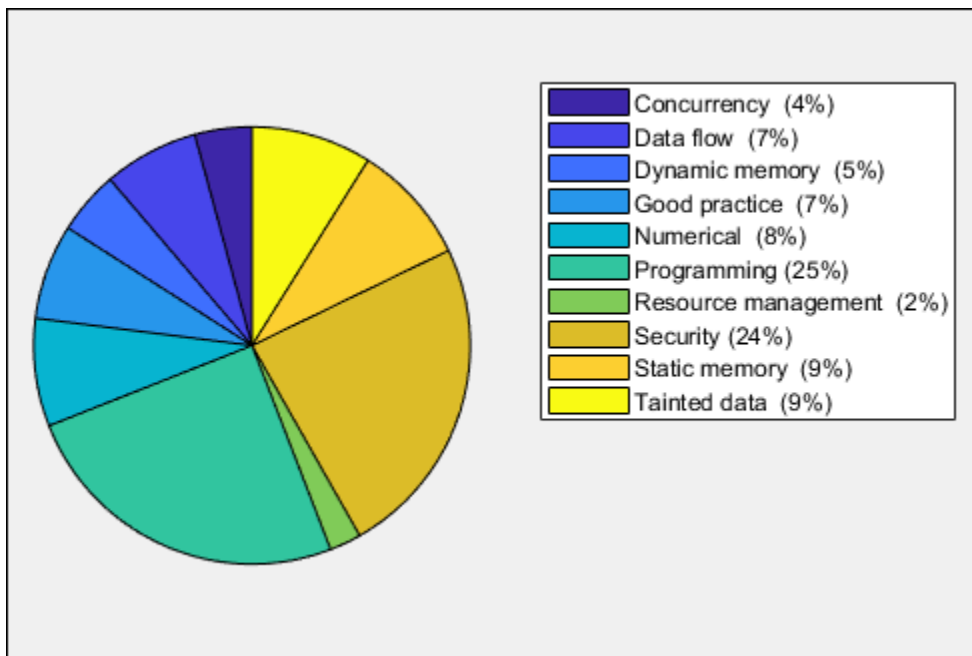
- `polyspace.BugFinderResults`: Read Bug Finder results into a table.

- `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 19-5
- “Export Polyspace Analysis Results to Excel by Using MATLAB Scripts” on page 19-7

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” on page 19-2.

- 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 3-2.

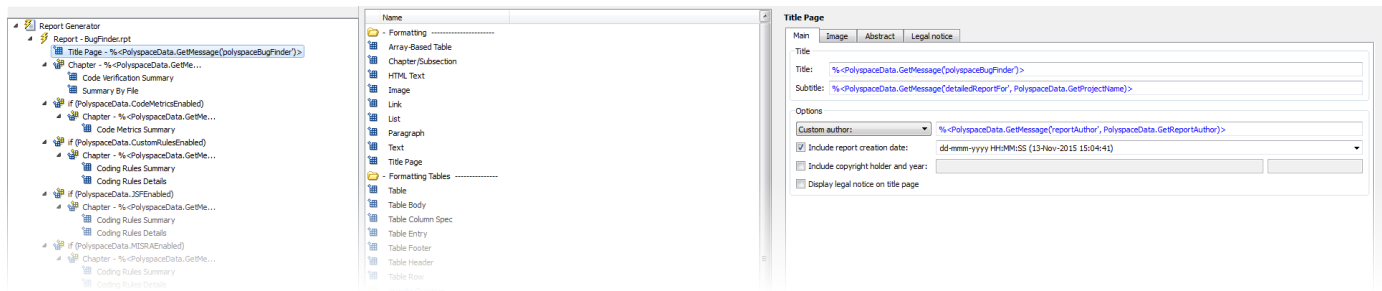
- 2 Open the Report Explorer interface. At the MATLAB command prompt, enter:

```
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.

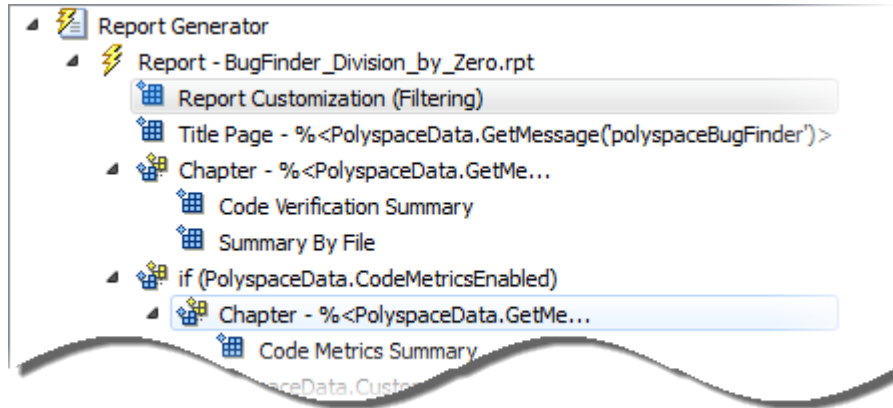
In this example, you add a component to the **BugFinder** template so that the template includes only **Integer division by zero** and **Float division by zero** defects in a report.

- 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.

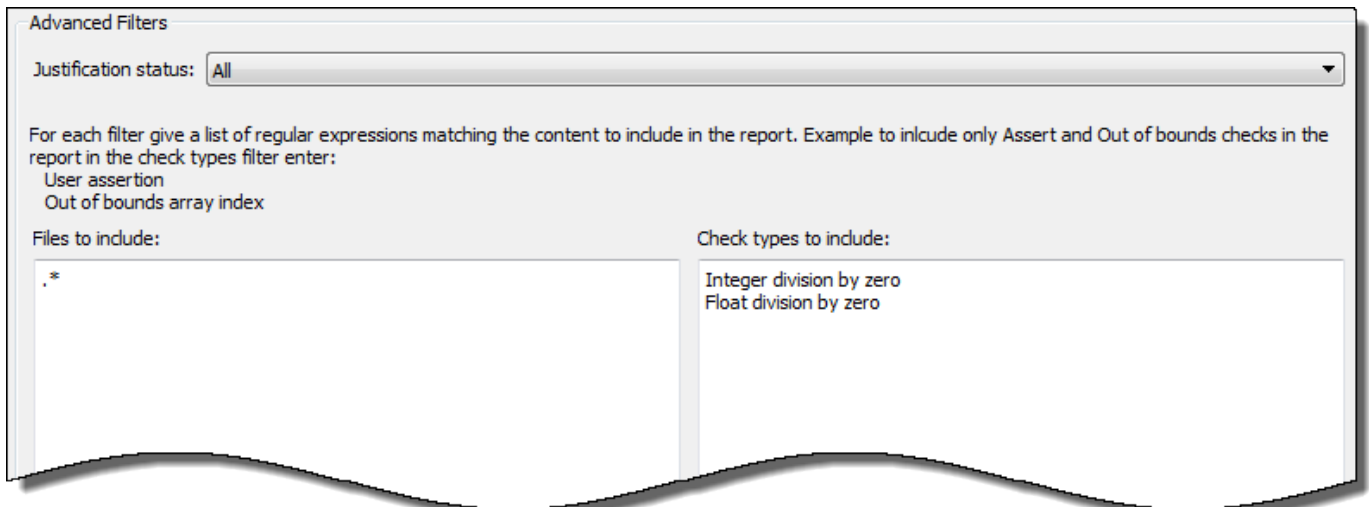
To perform this action:

- a Drag the component **Report Customization (Filtering)** from the middle pane and drop 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**.

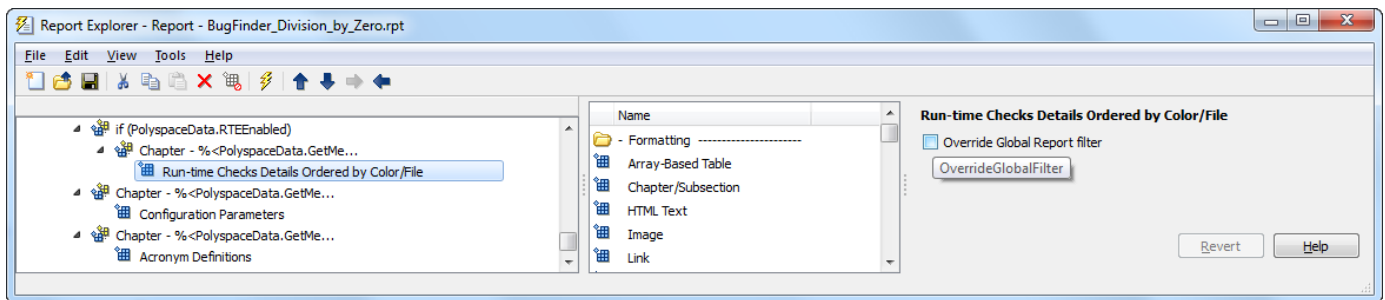
- 3 Change an existing chapter-specific component so that it does not override the global filter you applied in the previous step. If you prevent the overriding, the chapter-specific component follows the filtering specifications in the global component.

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.



- 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.psrj**.

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” on page 19-2.

In the report, open **Chapter 4. Defects**. *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 5. Defects**. *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.

Software Quality with Polyspace Metrics

- “Upload Results to Polyspace Metrics” on page 20-2
- “View Projects in Polyspace Metrics” on page 20-4
- “Compare Metrics Against Software Quality Objectives” on page 20-11
- “Web Browser Requirements for Polyspace Metrics” on page 20-19
- “View Results List in Polyspace Metrics” on page 20-20

Upload Results to Polyspace Metrics

Note This topic describes a workflow in the Polyspace Metrics web interface.

For easier collaborative reviews, use Polyspace Bug Finder Access™. In addition to a more intuitive web dashboard, with Polyspace Access you can:

- Review and justify results directly from your web browser.
- Integrate a defect-tracking tool such as Jira with the web interface and create tickets to track Polyspace findings.
- Share analysis results using web links.

For more information, see the Polyspace Bug Finder Access documentation.

After analysis, you can upload results to the Polyspace Metrics web interface. The web interface displays a summary of your analysis results. You can share this summary with others even if they do not have Polyspace installed locally. You can also compare the results against previous analyses on the same project or measure them against predefined software quality objectives.

For more information, see “Polyspace Metrics Interface” on page 20-7.

Before you generate code quality metrics, set up Polyspace Metrics. See “Set Up Polyspace Metrics”.

Manually Upload Results

To upload your results to the Polyspace Metrics web interface,

- 1 Select your results in the Project Browser pane.
- 2 Select **Metrics > Upload to Metrics**.
- 3 When you upload results to Polyspace Metrics, you are prompted to enter a password. Leave the field blank if you do not want to specify one.

If you specify a password, you must enter it every time you open your project in Polyspace Metrics. The session lasts for 30 minutes even if you close and reopen your web browser. After 30 minutes, enter your password again.

You can also specify a password later. On the Polyspace Metrics web interface, right-click your project and select **Change/Set Password**.

Note The password for a Polyspace Metrics project is encrypted. The web data transfer is not encrypted. The password feature minimizes unintentional data corruption, but it does not provide data security. However, data transfers between a Polyspace local host and the remote analysis MATLAB Job Scheduler host are always encrypted. To use a secure web data transfer with HTTPS, see “Configure Web Server for HTTPS” (Polyspace Code Prover).

If you upload results from multiple modules in a project, the results have the same project name and version number but appear under separate modules in Polyspace Metrics. To see or change the project name and version number, right-click a project in the **Project Browser** pane and select **Project Properties**.

Command Line: Use the command `polyspace-results-repository`. For a quick review of the command options, use the `-h` flag. At the command line, enter:

```
polyspaceroot\polyspace\bin\polyspace-results-repository -h
```

Here, *polyspaceroot* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

Automatically Upload Results (Batch Analysis Only)

If you perform a remote analysis, you can specify for the results to be uploaded automatically to the web interface after analysis. Otherwise, upload the results after analysis manually.

- 1 On the **Configuration** pane, select **Run Settings**.
- 2 Along with **Run Bug Finder analysis on a remote cluster**, select **Upload results to Polyspace Metrics**.

After analysis, the results are automatically uploaded to the web interface.

- 3 When you upload results to Polyspace Metrics, you are prompted to enter a password. Leave the field blank if you do not want to specify one.

If you specify a password, you must enter it every time you open your project in Polyspace Metrics. The session lasts for 30 minutes even if you close and reopen your web browser. After 30 minutes, enter your password again.

You can also specify a password later. On the Polyspace Metrics web interface, right-click your project and select **Change/Set Password**.

Note The password for a Polyspace Metrics project is encrypted. The web data transfer is not encrypted. The password feature minimizes unintentional data corruption, but it does not provide data security. However, data transfers between a Polyspace local host and the remote analysis MATLAB Job Scheduler host are always encrypted. To use a secure web data transfer with HTTPS, see “Configure Web Server for HTTPS” (Polyspace Code Prover).

If you upload results from multiple modules in a project, the results have the same project name and version number but appear under separate modules in Polyspace Metrics. To see or change the project name and version number, right-click a project in the **Project Browser** pane and select **Project Properties**.

Command Line: Use the option `Upload results to Polyspace metrics (-add-to-results-repository)`.

See Also

`polyspace-results-repository`

Related Examples

- “View Projects in Polyspace Metrics” on page 20-4
- “Compare Metrics Against Software Quality Objectives” on page 20-11

View Projects in Polyspace Metrics

Note This topic describes a workflow in the Polyspace Metrics web interface.

For easier collaborative reviews, use Polyspace Bug Finder Access . In addition to a more intuitive web dashboard, with Polyspace Access you can:

- Review and justify results directly from your web browser.
- Integrate a defect-tracking tool such as Jira with the web interface and create tickets to track Polyspace findings.
- Share analysis results using web links.

For more information, see the Polyspace Bug Finder Access documentation.

Polyspace Metrics is a web dashboard that displays code quality metrics from your analysis results. Using this dashboard, you can:

- Track improvements or regression in code quality over time.
- Provide managers a high-level overview of your code quality.
- Compare your code against quality objectives.
- Narrow your analysis review to critical results.

Upload Results

Before you can review your project in Polyspace Metrics, you must “Upload Results to Polyspace Metrics” on page 20-2.

Open Metrics Interface

You can open the metrics interface in one of the following ways:

- If you have Polyspace installed, select **Metrics > Open Metrics**.
- If you do not have Polyspace installed, open a web browser and enter the following URL:

protocol:// ServerName: PortNumber

- *protocol* is either `http` (default) or `https`.

To use HTTPS, additional configuration is required. See “Configure Web Server for HTTPS” (Polyspace Code Prover).

- *ServerName* is the name or IP address of your Polyspace Metrics server.
- *PortNumber* is the web server port number (default 8080).

When the Polyspace Metrics web interface opens you are presented with a list of results in your repository. You can view these results by project or by run.

The **Projects** tab lists the uploaded results by projects. On this tab, you can:

- See the number of project runs and overall statistics about the project by hovering your cursor over the project name.
- See project-level metrics by right-clicking the column headers and adding additional columns: Bug Finder Checks, Coding Rules, Code Metrics, Run-Time Errors, or Review Progress.
- Create project groups by right-clicking a project and selecting **Create Project Category**. Drag projects to your new category.
- Filter projects using the column headers.
- Delete projects from the Metrics repository by right-clicking the project and selecting **Delete Project from Repository**.
- Assign or change the password for a project by right-clicking the project and selecting **Change/Set Password**.
- Review into code quality metrics for a project by clicking the project. For details, see “Polyspace Metrics Interface” on page 20-7.

The **Runs** tab lists the individual runs for all projects. On this tab, you can:

- Delete a run from the repository by right-clicking the run and selecting **Delete Run from Repository**.
- Assign password to run by right-clicking the run and selecting **Change/Set Password**.
- See runs between two specific dates by selecting the starting date in the **From** field and the end date in the **To** field.
- See only the last n runs by changing the value of the **Maximum number of runs** field.
- See code quality metrics for a run by right-clicking the run and selecting **Go to Metrics Page**.
- Download results of run to Polyspace user interface by clicking the run name.

Review Metrics

For each project or analysis, you can view the code quality metrics spread over four tabs, at project, file, and function level. Select a project and you see four tabs:

- The **Summary** tab on page 20-7 provides a high-level overview of the verification results.
- The **Code Metrics** tab on page 20-8 provides the details of the code complexity metrics in your results.
- The **Coding Rules** tab on page 20-8 provides the details of the coding rule violations in your results.
- The **Bug Finder** tab on page 20-9 provides details of code defects in your results.

If you want to “Compare Metrics Against Software Quality Objectives” on page 20-11, before reviewing your results, you can turn on quality objectives.

- 1 Click an entry on the **Summary** tab. Clicking on an entry brings you to the respective tab for more details.
- 2 On the **Code Metrics**, **Coding Rules** or **Run-Time Errors** tabs, select an entry to download the result to the Polyspace user interface.

The results appear on the **Results List** pane in the Polyspace user interface. The filter **Show > Web checks** on this pane indicate that you have downloaded the results from Polyspace Metrics.

- 3 In the Polyspace user interface, review the particular result, investigate the root cause in your source code, and assign review comments and justifications.
- 4 To upload your comments and justifications to the Polyspace Metrics repository, select **Metrics > Upload to Metrics**.

Tip To upload automatically your comments and justifications to the Polyspace Metrics repository when you save them:

- a Select **Tools > Preferences**.
 - b On the **Server Configuration** tab, select **Save justifications in the Polyspace Metrics repository**.
-

- 5 In the Polyspace Metrics interface, click  to view your newly uploaded metrics.

Compare Metrics Between Results

Using the Polyspace Metrics interface, you can track improvements or regression in code quality metrics over various runs on the same source code.

To view trends in metrics, upload the various versions of your results to the Polyspace Metrics repository.

- 1 Open the Polyspace Metrics interface.

For more information, see “Open Metrics Interface” on page 20-4.




- 2 On the **Projects** tab, select the project for which you want to view trends.

The code quality metrics for all versions of the project appear on the **Summary**, **Code Metrics**, **Coding Rules**, and **Bug-Finder** tabs.

- 3 To compare two versions of the same project:

- a In the **From** and **To** lists on the upper left of the web dashboard, select the two versions that you want to compare.
- b Select the **Compare** box.

On each tab, new columns appear and existing columns display improvement or regression in a metric. For example, in the figure below, you see a new **All Metrics Trend** column that appears on the **Summary** tab. This column describes how the metrics in the **Bug-Finder** group compare over two versions of a project.

- A  means that the metrics is better.
- A  means that the metric is worse.
- A mixed  in the **All Metrics Trend** column means some metrics improved and some did not improve.

- 4 To see only the new findings in a version compared to a previous version:

- a In the **From** and **To** lists on the upper left of the web dashboard, select the two versions that you want to compare.
- b Select the **New Findings Only** box.

The existing columns display only the new findings. In addition, you also see two new columns:

- The **Newly Confirmed** column shows those new findings to which you assign a **Severity** of High, Medium, or Low in the Polyspace user interface.
- The **Newly Fixed** column shows those findings to which you had assigned a **Severity** of High, Medium, or Low in the previous run. However, the assignment does not exist in the current run, either because a red or orange check turned green, or because you changed the **Severity** to Unset.

Polyspace Metrics Interface

- “Summary Tab” on page 20-7
- “Code Metrics Tab” on page 20-8
- “Coding Rules Tab” on page 20-8
- “Bug-Finder Tab” on page 20-9

If you turn on Software Quality Objectives, each tab also specifies how your project or run compares against those objectives. See “Compare Metrics Against Software Quality Objectives” on page 20-11.

Summary Tab

The **Summary** tab summarizes the analysis results for a project or run.

Column Name		Description
Verification		Version number of the results and the source files.
Verification Status		Analysis level completed.
Code Metrics	Files	Number of files in project.
	Lines of code	Number of lines of code, broken down by file.
Coding Rules	Confirmed Defects	Number of coding rule violations assigned a Severity of High, Medium, or Low in the Polyspace user interface.
	Violations	Total number of coding rule violations. Statistics for AUTOSAR C++ 14, CERT C, CERT C++, and ISO/IEC TS 17961 coding standard violations are not available in the Polyspace Metrics interface.
Bug-Finder Checks	Confirmed Defects	Number of defects assigned a Severity of High, Medium, or Low in the Polyspace user interface.
	Checks	Total number of defects.
Software Quality Objectives	Overall Status	A status of PASS or FAIL based on whether your code satisfies the software quality objectives you specified.
	Level	The software quality objectives that you specify. You can either use a predefined set of objectives or specify your own objectives.
	Review Progress	Number or percent of justified results. To justify a result, you must assign a Status in the Polyspace user interface.

Column Name		Description
Verification		Version number of the results and the source files.
	Justified Code Metrics	Number or percent of code metric threshold violations that you have justified. To justify a result, you must assign a Status in the Polyspace user interface.
	Justified Coding Rules	Number or percent of coding rule violations that you have justified. To justify a result, you must assign a Status in the Polyspace user interface.
	Justified Bug-Finder Checks	Number or percent of defects that you have justified. To justify a result, you must assign a Status in the Polyspace user interface.

Code Metrics Tab

The **Code Metrics** tab lists the code complexity metrics for your project or run.

Some metrics are calculated at the project level, while others are calculated at file or function level. For metrics calculated at the function level, the entry displayed for a file is either an aggregate or a maximum over the functions in the file.

For more information, see “Code Metrics”.

Coding Rules Tab

The **Coding Rules** tab lists the coding rule violations in your project or run. For more information on the coding rules, see “Coding Standards”.

You cannot view results for these coding standards in the Polyspace Metrics interface.

- AUTOSAR C++ 14.
- CERT C.
- CERT C++.
- ISO/IEC TS 17961.

To review coding rule violations for these standards, use the Polyspace Bug Finder desktop interface or Polyspace Access (Polyspace Bug Finder Access).

You can group the information in the columns by **Files** or **Coding Rules**.

Column Name		Description
Coding Rules	Confirmed Defects	Number of coding rule violations assigned a Severity of High, Medium, or Low in the Polyspace user interface.

Column Name		Description
	Justified	Number of coding rule violations that you have justified. To justify a result, you must assign a Status in the Polyspace user interface.
	Violations	Total number of coding rule violations.
Software Quality Objectives	Quality Status	A status of PASS or FAIL based on whether your code satisfies the software quality objectives you specified.
	Level	The software quality objectives that you specify. You can either use a predefined set of objectives, or specify your own objectives.
	Review Progress	Number or percent of justified coding rule violations. To justify a result, you must assign a Status in the Polyspace user interface.

Bug-Finder Tab

The **Bug-Finder** tab lists the “Defects” in your project or run.

You can group the information in the columns by **Files** or **Bug-Finder Checkers**.

Column Name		Description
Confirmed Defects		Number or percent of defects assigned a Severity of High, Medium, or Low in the Polyspace user interface. See “Address Polyspace Results Through Bug Fixes or Justifications” on page 17-2.
Bug-Finder Checks	Justified	Number or percent of justified defects. To justify a result, you must assign a Status in the Polyspace user interface.
	Checks	Total number of checks.
	High	Total number of “High Impact Defects” on page 18-9.
	Medium	Total number of “Medium Impact Defects” on page 18-11.
	Low	Total number of “Low Impact Defects” on page 18-16.
Software Quality Objectives	Quality Status	A status of PASS or FAIL based on whether your code satisfies the software quality objectives you specified.
	Level	The software quality objectives that you specify. You can either use a predefined set of objectives or specify your own objectives.
	Review Progress	Number or percent of justified defects. To justify a result, you must assign a Status in the Polyspace user interface.

See Also

Related Examples

- “Address Polyspace Results Through Bug Fixes or Justifications” on page 17-2
- “Upload Results to Polyspace Metrics” on page 20-2
- “Compare Metrics Against Software Quality Objectives” on page 20-11
- “Compare Metrics Between Results” on page 20-6

Compare Metrics Against Software Quality Objectives

Note This topic describes a workflow in the Polyspace Metrics web interface.

For easier collaborative reviews, use Polyspace Bug Finder Access . In addition to a more intuitive web dashboard, with Polyspace Access you can:

- Review and justify results directly from your web browser.
- Integrate a defect-tracking tool such as Jira with the web interface and create tickets to track Polyspace findings.
- Share analysis results using web links.

For more information, see the Polyspace Bug Finder Access documentation.

After generating and viewing metrics from your analysis results, you can review the results in greater detail.

To focus your review, you can:

- 1 Define quality objectives that you or developers in your organization must meet.
- 2 Apply the quality objectives to your analysis results.
- 3 Review only those results that fail to meet those objectives.

Apply Predefined Objectives to Metrics

By default, the software quality objectives are turned off. To apply quality objectives:


- 1 In the Polyspace Metrics interface, open the metrics page for a project.
- 2 From the **Quality Objectives** list in the upper left, select **ON**.

A new group of **Software Quality Objectives** columns appears.

- The **Overall Status** column shows the last used quality objective level to generate a status of **PASS** or **FAIL** for your results.
- The **Level** column shows the quality objective level.

To change your quality objective level, in this column, select a cell. From the drop-down list, select a quality level. For more information, see “Bug Finder Quality Objective Levels” on page 20-12.


- 3 For files with an **Overall Status** of **FAIL**, to see what causes the failure, view the entries in the other **Software Quality Objectives** columns. The failing levels are marked red.

If the  icon appears next to the status, it means that Polyspace does not have enough information to compute the status. For instance, if you specify BF-Q0-1, certain coding rules must be review. But, if you do not check coding rules during the analysis, Polyspace cannot determine whether your project satisfies the coding rule objectives specified in BF-Q0-1.

- 4 To investigate the failing quality objectives, select the entries marked red for more details.
- 5 On the **Code Metrics**, **Coding Rules**, or **Bug-Finder** tab,

- a Select the red column entries to download the results.
- b Review the violations and fix or justify the results.

See “Address Polyspace Results Through Bug Fixes or Justifications” on page 17-2.

- c Upload your new justifications to the Polyspace Metrics web dashboard.
- 6 After your review, in the Polyspace Metrics interface, click  to view the updated metrics.

If you change your code, to update the metrics, rerun your analysis and upload the results to the Polyspace Metrics repository. If you have justifications in your previous results, import them to the new results before uploading to the repository.

Bug Finder Quality Objective Levels

The Bug Finder Quality Objectives or BF-QOs are a set of thresholds against which you can compare your Bug Finder analysis results. These objectives are adapted from the Polyspace Code Prover “Software Quality Objectives” (Polyspace Code Prover). You can develop a review process based on the Quality Objectives.

You can use a predefined BF-QO level or define your own. 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"> • $n1$ — Number of different operators • $N1$ — Total number of operators • $n2$ — Number of different operands • $N2$ — 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” on page 18-9	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” on page 18-11	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" on page 18-16	0

BF-QO Exhaustive

In addition to all the requirements of BF-QO Level 1, 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

Customize Software Quality Objectives

Instead of using a predefined objective, you can define your own quality objectives and apply them to your project.

- 1 Save the following content in an XML file. Name the file `Custom-SQO-Definitions.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<MetricsDefinitions>

  <SQO ID="Custom-BF-QO-Level" ApplicableProduct="Bug Finder"
    ApplicableProject="My_Project">

    <comf>20</comf>
    <path>80</path>
    <goto>0</goto>
    <vg>10</vg>
    <calling>5</calling>
    <calls>7</calls>
    <param>5</param>
    <stmt>50</stmt>
```

```

<level>4</level>
<return>1</return>
<vocf>4</vocf>
<ap_cg_cycle>0</ap_cg_cycle>
<ap_cg_direct_cycle>0</ap_cg_direct_cycle>
<Num_Unjustified_Violations>Custom_MISRA_Rules_Set
    </Num_Unjustified_Violations>
<Num_Unjustified_BF_Checks>Custom_BF_Checks_Set
    </Num_Unjustified_BF_Checks>
</SQ0>

<CodingRulesSet ID="Custom_MISRA_Rules_Set">
  <Rule Name="MISRA_C_5_2">0</Rule>
  <Rule Name="MISRA_C_17_6">0</Rule>
</CodingRulesSet>

<BugFinderChecksSet ID="Custom_BF_Checks_Set">
  <Check Name="UNREACHABLE">0</Check>
  <Check Name="USELESS_IF">0</Check>
</BugFinderChecksSet>

</MetricsDefinitions>

```

You can use this file for both Bug Finder and Code Prover results. For information on the XML elements specific to Code Prover, see “Compare Metrics Against Software Quality Objectives” (Polyspace Code Prover).

- 2 Save this XML file in the folder where remote analysis data is stored, for example, C:\Users\JohnDoe\AppData\Roaming\Polyspace_RLData.s.

If you want to change the folder location, select **Metrics > Metrics and Remote Server Settings**.

- 3 To make the quality level Custom-BF-Q0-Level applicable to a certain project, replace the value of the ApplicableProject attribute with the project name.

If you want the quality objectives to apply to all projects, use ApplicableProject="".

- 4 Modify the file to specify your thresholds.

For specifying coding rules, begin the rule name with the appropriate string followed by the rule number. Use _ instead of a decimal point in the rule number. To specify directives, begin specifying the directive number with D_, for instance, MISRA_C3_D4_6, for MISRA C: 2012 directive 4.6.

Rule	String	Rule numbers
MISRA C: 2004	MISRA_C_	“MISRA C:2004 and MISRA AC AGC Coding Rules” on page 13-3
MISRA C: 2012	MISRA_C3_	“MISRA C:2012 Directives and Rules”
MISRA C++	MISRA_Cpp_	“MISRA C++:2008 Rules”
JSF C++	JSF_Cpp_	“JSF AV C++ Coding Rules” on page 13-48
Custom coding rules	Custom_	“Custom Coding Rules”

For specifying defects, use the defect acronym. See “Short Names of Bug Finder Defect Checkers” on page 14-26.

For specifying code metrics, use the code metric acronym. See “Short Names of Code Complexity Metrics” on page 17-13.

- 5 After you have made your modifications, in the Polyspace Metrics interface, open the metrics for your project. From the **Quality Objectives** list in the upper left, select **ON**.
- 6 On the **Summary** tab, select an entry in the **Level** column. For the project name that you specified, your new quality objective **Custom-BF-QO-Level** appears in the drop-down list.
- 7 Select your new quality objective.

The software compares the thresholds you had specified against your results and updates the **Overall Status** column with **PASS** or **FAIL**.

- 8 To define another set of custom quality objectives, add the following content to the Custom-BF-QO-Definitions.xml file:

```
<SQ0 ID="Custom-BF-QO-Level_2" ParentID="Custom-BF-QO-Level"
      ApplicableProduct="Bug Finder" ApplicableProject="My_Project">
  ...
</SQ0>
```

- ID represents the name of the new set.

You cannot have the same values of ID and ApplicableProject for two different sets of quality objectives. For example, if you use ID="Custom-BF-QO-Level" for two different custom sets, and ApplicableProject is either My_Project or "" for both sets, you see the following error:

```
The SQ0 level 'Custom-BF-QO-Level' is multiply defined.
```

- ParentID specifies another level from which the current level inherits its quality objectives. In the preceding example, the level Custom-BF-QO-Level_2 inherits its quality objectives from the level Custom-BF-QO-Level.

If you do not want to inherit quality objectives from another level, omit this attribute.

- . . . represents the additional quality thresholds that you specify for the level Custom-BF-QO-Level_2.

The quality thresholds that you specify override the thresholds that Custom-BF-QO-Level_2 inherits from Custom-BF-QO-Level. For instance, if you specify <goto>1</goto>, this objective overrides the threshold specification <goto>0</goto> of Custom-BF-QO-Level.

Elements in Custom Quality Objective Files

- “HIS Metrics” on page 20-18
- “Non-HIS Metrics” on page 20-18

The following tables list the XML elements that can be added to the custom BF-QO file. The content of each element specifies a threshold against which the software compares analysis results. For each element, the table lists the metric to which the threshold applies. Here, HIS refers to the Hersteller Initiative Software.

HIS Metrics

Element	Metric
comf	Comment Density (Polyspace Code Prover)
path	Number of Paths (Polyspace Code Prover)
goto	Number of Goto Statements (Polyspace Code Prover)
vg	Cyclomatic Complexity (Polyspace Code Prover)
calling	Number of Calling Functions (Polyspace Code Prover)
calls	Number of Called Functions (Polyspace Code Prover)
param	Number of Function Parameters (Polyspace Code Prover)
stmt	Number of Instructions (Polyspace Code Prover)
level	Number of Call Levels (Polyspace Code Prover)
return	Number of Return Statements (Polyspace Code Prover)
vocf	Language Scope (Polyspace Code Prover)
ap_cg_cycle	Number of Recursions (Polyspace Code Prover)
ap_cg_direct_cycle	Number of Direct Recursions (Polyspace Code Prover)
Num_Unjustified_Violations	Number of unjustified violations of coding rules specified by entries under the element CodingRulesSet
Num_Unjustified_BF_Checks	Number of unjustified defects of types specified by entries under the element BugFinderChecksSet

Non-HIS Metrics

Element	Description of metric
fco	Estimated Function Coupling (Polyspace Code Prover)
flin	Number of Lines Within Body (Polyspace Code Prover)
fxln	Number of Executable Lines (Polyspace Code Prover)
ncalls	Number of Call Occurrences (Polyspace Code Prover)

Web Browser Requirements for Polyspace Metrics

To use Polyspace Metrics, install Java®, version 1.4 or later on your computer.

Polyspace Metrics supports these web browsers:

- Internet Explorer® version 7.0, or later
- Firefox® version 3.6, or later
- Google® Chrome version 12.0, or later
- Safari for Mac version 6.1.4 and 7.0.4

Additional Considerations

- With certain web browsers such as Google Chrome, you are prompted to download or open a file called `psapplet.jsp`. You must configure your system to open `.jsp` files using the Java Web Start Launcher (`javaws`) binary. You can find this binary in your Java installation folder.

For instance, if you download `psapplet.jsp` on a Windows system, to open and view your Polyspace results:

- 1 Right-click `psapplet.jsp` and select **Properties**.
 - 2 Next to **Opens With**: select the Java Web Start Launcher (`javaws`) binary. If the binary is not in the default list of options, browse to your Java installation folder, for example `C:\Program Files\Java\jre1.8.0_161\bin`.
 - 3 After you apply your changes, double-click `psapplet.jsp` to open it.
- If your computer uses the Linux operating system, manually install the required Java plug-in for the Firefox web browser:
 - 1 Create a `plugins` folder under the `.mozilla` folder in your home directory:

```
mkdir ~/.mozilla/plugins
```
 - 2 From this folder, create a symbolic link to the Java plug-in, which is available in the Java Runtime Environment folder.

For instance, to use the Java Runtime Environment of your MATLAB installation, enter these commands:

```
cd ~/.mozilla/plugins
ln -s MATLAB_Install/sys/java/jre/glnxa64/jre/lib/amd64/libnjp2.so
```

- If you download results using Internet Explorer 11, it may take a minute or two to open the Java plug-in and load the Polyspace interface.

View Results List in Polyspace Metrics

Note This topic describes a workflow in the Polyspace Metrics web interface.

For easier collaborative reviews, use Polyspace Bug Finder Access . In addition to a more intuitive web dashboard, with Polyspace Access you can:

- Review and justify results directly from your web browser.
- Integrate a defect-tracking tool such as Jira with the web interface and create tickets to track Polyspace findings.
- Share analysis results using web links.

For more information, see the Polyspace Bug Finder Access documentation.

This example shows how to use Polyspace Metrics to view the Results List in and download results. Your results appear in Polyspace Metrics if,

- Before analyzing your code in batch mode, you selected the **Add to results repository** analysis option.
- After analyzing your code, batch or local mode, you selected **Metrics > Upload to Metrics**.

Open Polyspace Metrics

- 1 From the Polyspace interface, select **Metrics > Open Metrics**.

You can also open the Polyspace Metrics Web UI using the URL:

protocol://ServerName:PortNumber


- *protocol* is either http (default) or https.

To use HTTPS, you must configure the web server for HTTPS.



- *ServerName* is the name or IP address of your Polyspace Metrics server.
- *PortNumber* is the web server port number (default 8080).

On the Metrics homepage, you can see all projects uploaded to your Polyspace Metrics repository.

Polyspace® Metrics

From To Maximum number of projects Refresh 

Projects | **Runs**

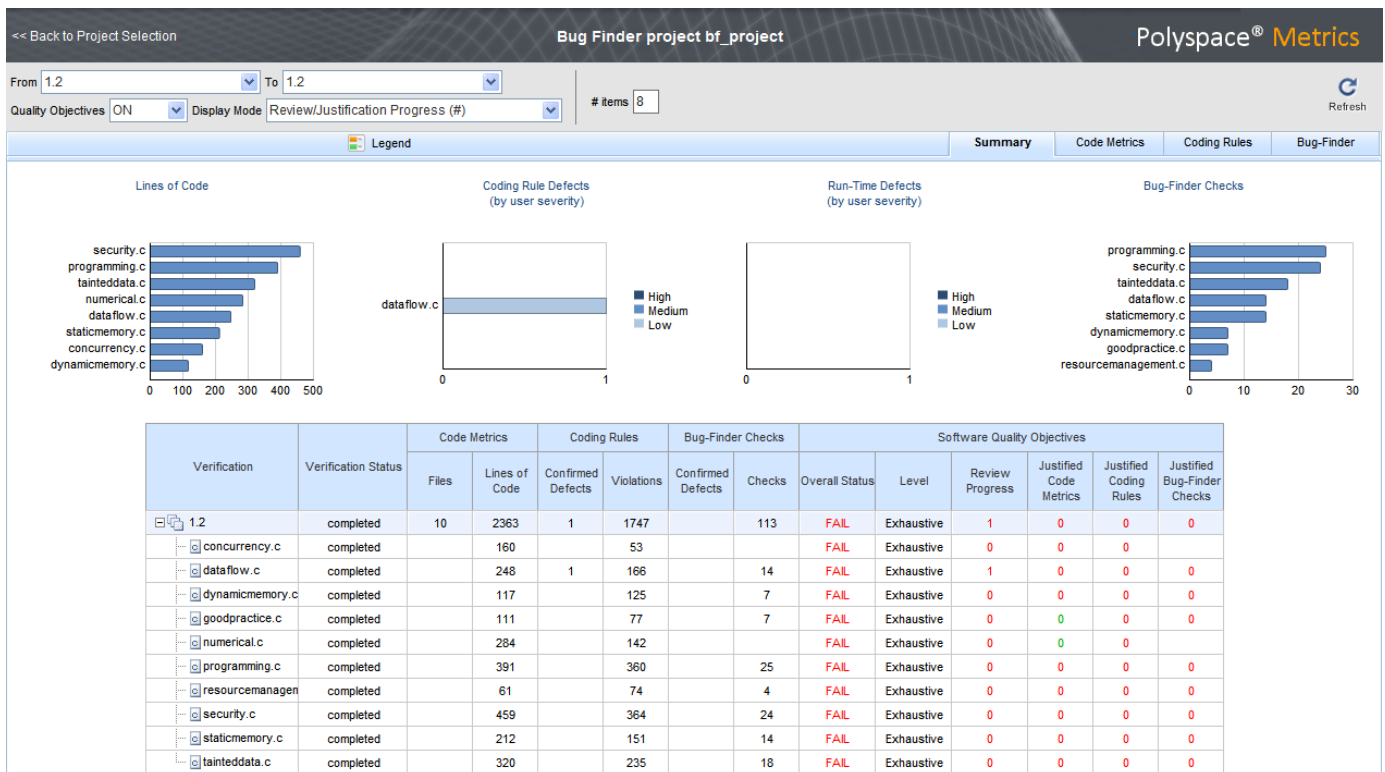
bf_project		Project	Product	Mode	Language	Latest Version	Date	Status
Language	C	<input type="text"/>	<input type="text" value=""/>	<input type="text" value=""/>	<input type="text" value=""/>	<input type="text" value=""/>	<input type="text" value=""/>	<input type="text" value=""/>
Mode		 bf_project	Bug Finder		C	1.2	Dec 01, 2015	completed
Last Run Name	1.2	 cp_project	Code Prover	Integration	C,C++	1.2	Dec 01, 2015	completed (PASS2)
Number of Runs	3							
Code Metrics								
Files	10							
Lines Of Code	2363							
Coding Rules								
Confirmed Defects	1							
Violations	1747							
Bug-Finder Checks								
Confirmed Defects								
Checks	113							
New Findings	113							
Review Progress	0.1%							

This site is optimized for Internet Explorer 7.0 (and above), Firefox 3.6 (and above) and Google Chrome 12.0 (and above).

View Results List

- 1 Select the **Projects** tab.
- 2 Hover over the project name to see a summary of the project results.
- 3 To see more details, select the project name.

The project opens to a Results List for the project.



Polyspace Metrics shows the summary graphically

Confirmed Defects column lists the number of coding rule violations or checks that you have reviewed and assigned a **Severity** of High, Medium, or Low in the Polyspace user interface.

4 To view the results in more detail, select the tabs:

- **Code Metrics:** Statistics about your project such as number of lines, header files, and function calls. To see code metrics, you must enable the analysis option Calculate code metrics (-code-metrics).
- **Coding Rules:** Description of coding rule violations.

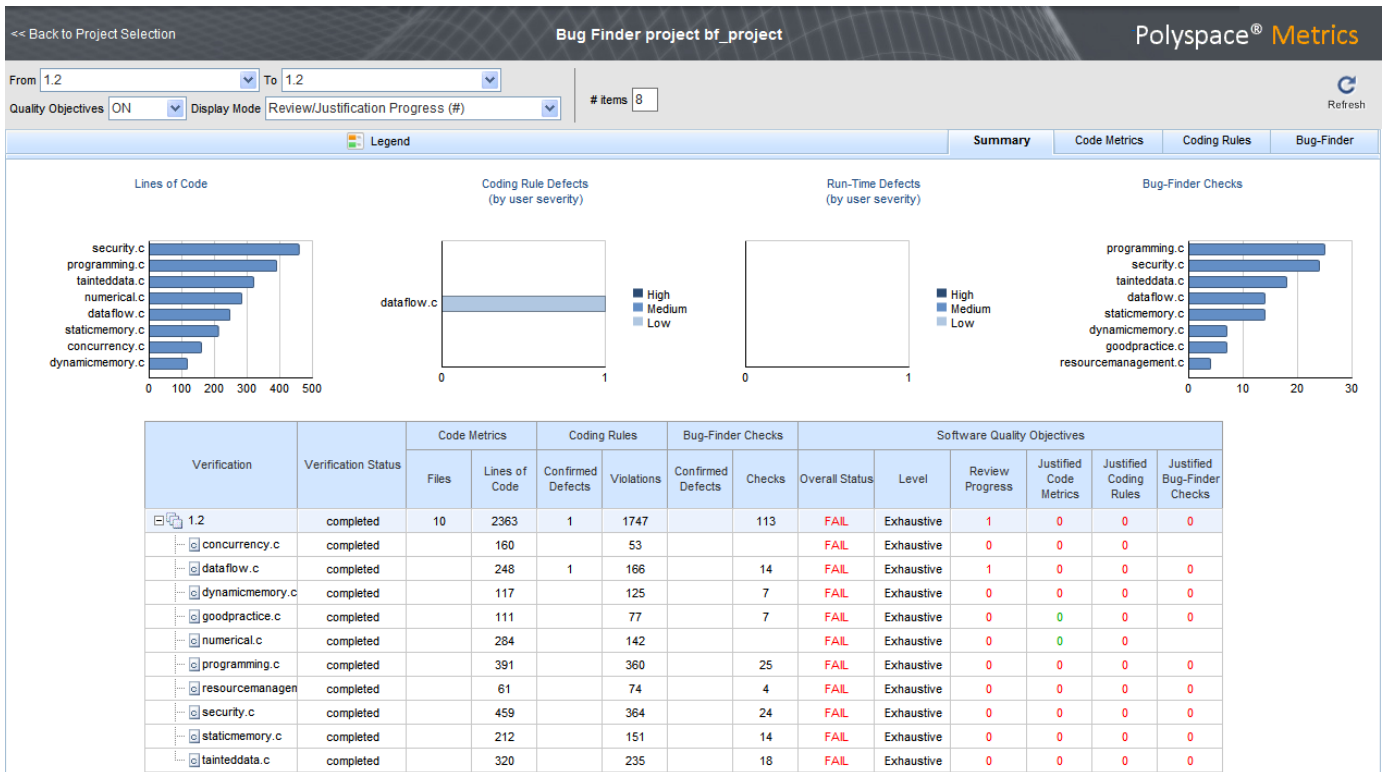
Statistics for AUTOSAR C++ 14, CERT C, CERT C++, and ISO/IEC TS 17961 coding standard violations are not available in the Polyspace Metrics interface.

- **Bug-Finder:** Description of defects detected by Polyspace Bug Finder.

Download Results

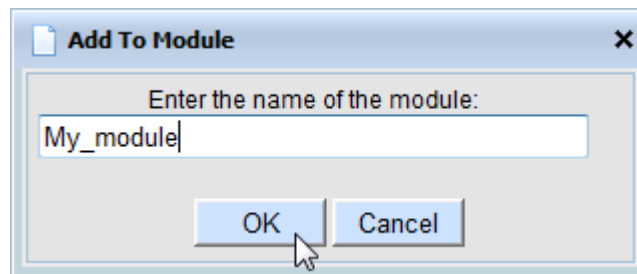
- 1 Select the **Projects** tab.
- 2 To view the Results List for your project, on the **Projects** column, select the project name.

The Results List for the project appears on the web page under the **Summary** tab.



3 To download results:

- Individual file — click a file name in the **Verification** column.
- Whole project — click a version number in the **Verification** column.
- Group of files —
 - a Right-click the row containing a file in the group. From the context menu, select **Add To Module**.
 - b Enter the name of your module in the dialog box. Click **OK**.



The name of the module appears on the **Verification** column.

- c Drag and drop the other files in the group to the module.
- d Click the name of the module and follow any prompt from your web browser. If the results do not open automatically in the Polyspace interface, check the “Web Browser Requirements for Polyspace Metrics” on page 20-19.

The results open on the **Results List** pane in Polyspace Bug Finder. The filter **Show > Web checks** on this pane indicate that you have downloaded the results from Polyspace Metrics.

You can review the downloaded results or generate reports. If you generate a report, all results in the results file appear in the report (not just the downloaded results). To generate a filtered report, change the scope **Web checks** to another named filtered set, for instance, **All results**. Then, apply filters and generate the report. For more information, see “Generate Reports” on page 19-2.

See Also

Related Examples

- “Set Up Polyspace Metrics”
- “Address Polyspace Results Through Bug Fixes or Justifications” on page 17-2

Troubleshooting in Polyspace Bug Finder

- “License Error -4,0” on page 21-2
- “View Error Information When Analysis Stops” on page 21-3
- “Contact Technical Support About Issues with Running Polyspace” on page 21-6
- “Compiler Not Supported for Project Creation from Build Systems” on page 21-9
- “Slow Build Process When Polyspace Traces the Build” on page 21-15
- “Check if Polyspace Supports Build Scripts” on page 21-16
- “Troubleshooting Project Creation from MinGW Build” on page 21-18
- “Troubleshooting Project Creation from Visual Studio Build” on page 21-19
- “Polyspace Cannot Find the Server” on page 21-20
- “Job Manager Cannot Write to Database” on page 21-21
- “Undefined Identifier Error” on page 21-22
- “Unknown Function Prototype Error” on page 21-25
- “Error Related to #error Directive” on page 21-26
- “Large Object Error” on page 21-27
- “Errors Related to Generic Compiler” on page 21-29
- “Errors Related to Keil or IAR Compiler” on page 21-30
- “Errors Related to Diab Compiler” on page 21-31
- “Errors Related to Green Hills Compiler” on page 21-33
- “Errors Related to TASKING Compiler” on page 21-35
- “Errors from Conflicts with Polyspace Header Files” on page 21-37
- “Errors from Using Namespace std Without Prefix” on page 21-38
- “Errors from Assertion or Memory Allocation Functions” on page 21-39
- “Errors from In-Class Initialization (C++)” on page 21-40
- “Errors from Double Declarations of Standard Template Library Functions (C++)” on page 21-41
- “Errors Related to GNU Compiler” on page 21-42
- “Errors Related to Visual Compilers” on page 21-43
- “Eclipse Java Version Incompatible with Polyspace Plug-in” on page 21-45
- “Coding Standard Violations Not Displayed” on page 21-46
- “Insufficient Memory During Report Generation” on page 21-48
- “Error or Slow Runs from Disk Defragmentation and Anti-virus Software” on page 21-49
- “SQLite I/O Error” on page 21-51
- “Errors with Temporary Files” on page 21-52
- “Resolve -xml-annotations-description Errors” on page 21-54

License Error -4,0

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

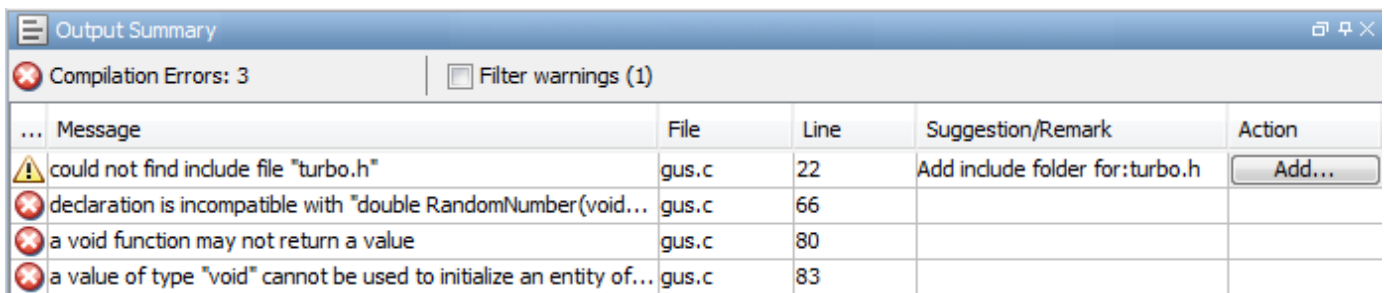
- 1 View the errors on the **Output Summary** tab.

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.
- 3 If you enable the Compilation Assistant, to fix an error, you can perform certain actions on the **Output Summary** tab.

The following figure shows an error due to a missing include file `turbo.h`. You can add the missing file by clicking the **Add** button on the **Output Summary** tab.



To turn on the Compilation Assistant, select **Tools > Preferences**. On the **Project and Results Folder** tab, select **Use Compilation Assistant**.

Note the following:

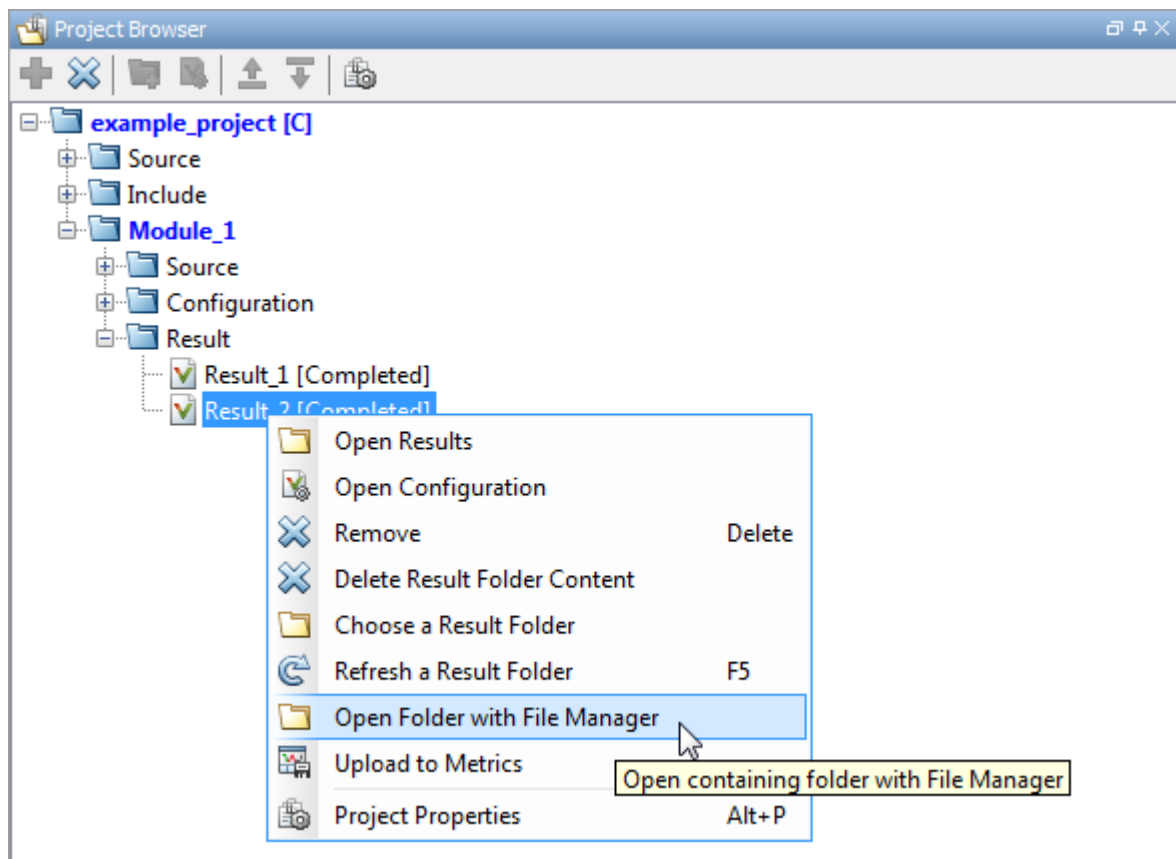
- By default, if some files do not compile, Bug Finder analyzes the remaining files. If you turn on Compilation Assistant, all files must compile. You do not get analysis results even if there is a single compilation error.
- The Compilation Assistant is disabled if you specify the option `Command/script` to apply to preprocessed files (`-post-preprocessing-command`)

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

File does not compile | Stop analysis if a file does not compile (-stop-if-compile-error)

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, run the following command, replacing *polyspaceroot* with your Polyspace installation folder:
 - UNIX — `polyspaceroot/polyspace/bin/polyspace-code-prover -ver`
 - Windows — `polyspaceroot\polyspace\bin\polyspace-code-prover -ver`

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.

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 “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 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.

Compiler Not Supported for Project Creation from Build Systems

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 9-20.

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 21-13.
- 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
Diab	Wind River	diab.xml
gcc	Not applicable	gcc.xml
Green Hills	Green Hills Software	ghs_arm.xml
		ghs_arm64.xml
		ghs_i386.xml
		ghs_ppc.xml

Compiler Name	Vendor	XML File
		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-rl78.xml
Renesas	Renesas	renesas-rh850.xml
		renesas-rl78.xml
		renesas-rx.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	xc8.xml

Slow Build Process When Polyspace Traces the 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, do one of the following:

- Launch the Cygwin shell from `cmd.exe` and then run your build script. For instance, 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" -c build.sh
```

- Find the full path to your build script and then run this script from `cmd.exe`.

For instance, enter the following command at the DOS command line:

```
cmd.exe /C path_to_script
```

`path_to_script` is the full path to your build script. For instance, `C:\my_scripts\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" -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\R2017b\polyspace\bin\polyspace-configure.exe"  
-output-options-file myOptions.txt launching.bat
```

You can now run `polyspace-bug-finder` on the options file.

Troubleshooting 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__))"
```

Troubleshooting Project Creation from Visual Studio Build

You can run `polyspace-configure` on a Visual Studio build and extract information from the build to create a Polyspace 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 Project Using Visual Studio Information” on page 1-14.

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`

Polyspace Cannot Find the 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”.

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)

Undefined Identifier Error

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 User Interface” on page 1-2.

- 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 9-28.

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.

Unknown Function Prototype Error

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 User Interface” on page 1-2.

- At the command line, use the flag `-I` with the `polyspace-bug-finder` command.

For more information, see `-I`.

Error 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)`.

Large Object Error

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.

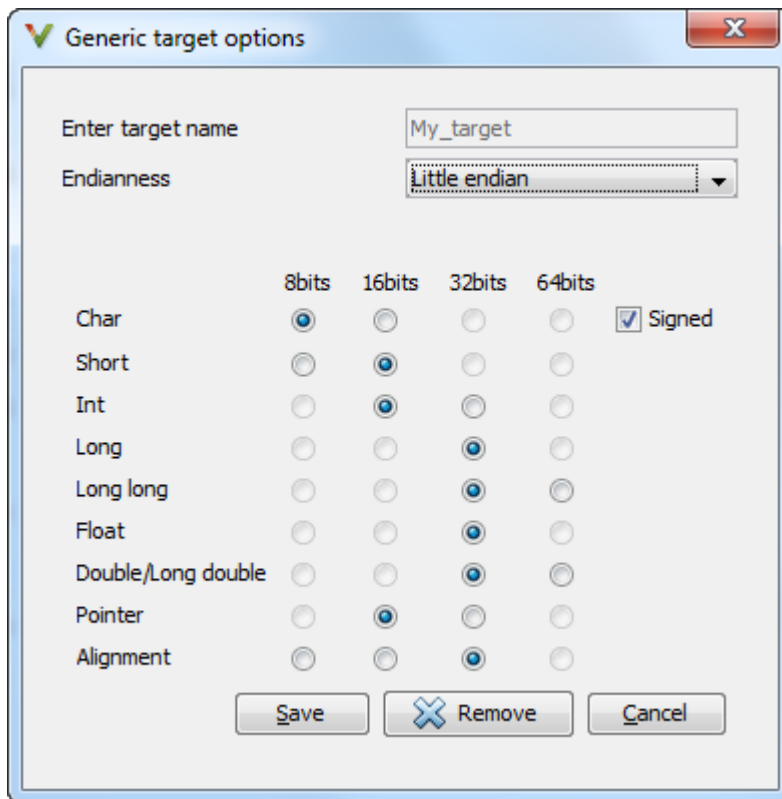
For instance, you get the message, `Limitation: struct or union is too large` in the following example. You specify a pointer size of 16 bits. The maximum object size allocated to a pointer, and therefore the maximum allowed size for an object, can be $2^{16}-1$ bytes. However, you declare a structure 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.

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.

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 9-23.

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)`.

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 compiler flag.

```
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;
}
```

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.
-

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 9-20.

Errors from Conflicts with Polyspace Header Files

Issue

You see compilation errors from header files included by Polyspace.

For instance, the error message refers to one of the subfolders of *polyspaceroot*\polyspace\verifier\cxx\include.

Typically, the error message is related to a standard library function.

Cause

If your compiler defines a standard library function or another construct and you do not provide the path to your compiler header files, Polyspace uses its own implementation of the function.

If your compiler definitions differ from the corresponding Polyspace definitions, the verification stops with an error.

Solution

Specify the folder containing your compiler header files.

- In the user interface, add the folder to your project.

For more information, see “Add Source Files for Analysis in Polyspace User Interface” on page 1-2.

- At the command line, use the flag `-I` with the `polyspace-bug-finder` 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 9-19.

Errors from Using 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**.

Errors from 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.

Errors from 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
class Test { public: static int m_number; };	int Test::m_number = 0;

Errors from Double Declarations of Standard Template Library Functions (C++)

Consider the following code.

```
#include <list>

void f(const std::list<int*>::const_iterator it) {}
void f(const std::list<int*>::iterator it) {}
void g(const std::list<int*>::const_reverse_iterator it) {}
void g(const std::list<int*>::reverse_iterator it) {}
```

The declared functions belong to `list` container classes with different iterators. However, the software generates the following compilation errors:

```
error: function "f" has already been defined
error: function "g" has already been defined
```

You would also see the same error if, instead of `list`, the specified container was `vector`, `set`, `map`, or `deque`.

To avoid the double declaration errors, do one of the following:

- Deactivate automatic stubbing of standard template library functions. For more information, see `No STL stubs (-no-stl-stubs)`.
- Define the following Polyspace preprocessing directives:
 - `__PST_STL_LIST_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_VECTOR_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_SET_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_MAP_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_DEQUE_CONST_ITERATOR_DIFFER_ITERATOR__`

For example, for the given code, run analysis at the command line with the following flag. The flag defines the appropriate directive for the `list` container.

```
-D __PST_STL_LIST_CONST_ITERATOR_DIFFER_ITERATOR__
```

For more information on defining preprocessor directives, see `Preprocessor definitions (-D)`.

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`.

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
```


Eclipse Java Version Incompatible with Polyspace Plug-in

In this section...

"Issue" on page 21-45

"Cause" on page 21-45

"Solution" on page 21-45

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 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.

Coding Standard Violations Not Displayed

Issue

You expect a coding rule violation on a line of code but the Polyspace analysis does not show the violation.

Possible Cause: Rule Checker Not Enabled

You might be looking for a reduced subset of coding rules.

For instance, if you check for MISRA C: 2012 rules, by default, the analysis looks for the mandatory-required subset only.

Solution

Check the coding rules options that you use. See:

- Check MISRA C:2004 (-misra2)
- Check MISRA C:2012 (-misra3)
- Check MISRA C++:2008 (-misra-cpp)
- Check JSF AV C++ rules (-jsf-coding-rules)

Possible Cause: Rule Violations in Header Files

All coding rule violations in the file might be suppressed.


For instance, by default, coding rule violations are suppressed from header files that are not in the same location as the source files.


Solution

Check the files where you suppress coding rule violations. See `Do not generate results for (-do-not-generate-results-for)`.

Possible Cause: Rule Violations in Macros

The rule violation occurs in a macro expansion. To save you from reviewing the same violation multiple times, the violation is shown on the macro definition instead of the macro usage. If the definition occurs in a header file, it might be suppressed from the results.

On the **Source** pane, you can tell if a line contains a macro expansion. Look for the  icon.

```
110  s8_ret = (s8) NA_VALUE;
```

Solution

Find the macro definition and see if it occurs in a header file. Determine if you are suppressing coding rule violations from header files. See `Do not generate results for (-do-not-generate-results-for)`.

Possible Cause: Compilation Errors

If any source file in the analysis does not compile, coding rules checking will be incomplete. The coding rules checker results:

- May not contain full results for files that did not compile
- May not contain full results for the files that did compile as some rules are checked only after compilation is complete

Check for compilation errors. See “View Error Information When Analysis Stops” on page 21-3.

Note When you enable the Compilation Assistant *and* coding rules checking, the software does not report coding rule violations if there are compilation errors.

Possible Cause: Code Prover Analysis with Lower Verification Level

If you run a Code Prover analysis to source compliance checking using the option `Verification level (-to)`, you might not see violations of some rules. These rules are checked in the later stages of a Code Prover analysis.

This reasoning applies to specific rules and does not apply to a Bug Finder analysis. See “Check for Coding Standard Violations” on page 12-2.

Solution

If you do not see a violation of one of those rules, check if your Code Prover analysis runs up to source compliance checking only. Use a higher value for the option `Verification level (-to)`.

Insufficient Memory During Report Generation

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 Cause

During generation of very large reports, the software can sometimes indicate that there is insufficient memory.

Solution

If this error occurs, 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*.

Error or Slow 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 whitelist 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 ps_ in the subfolder bin\win64.

- Configure the anti-virus software to exclude your temporary folder, for example, C:\Temp, from the checking process.

See “Storage of Temporary Files” on page 1-13.

SQLite I/O Error

Issue

When you try to run Polyspace, you get this error message:

Cause

Polyspace uses an SQLite database for storing results. This error can appear when SQLite databases are saved on NFS (Network File System) folders.

Solution

Check the folder where you save Polyspace results. For instance, if you run Polyspace at the command line, check the option `-results-dir`.

If the folder is an NFS folder, use a local folder instead.

Errors with 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” on page 1-13.

- 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” on page 1-13.

- 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” on page 1-13.

- 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.

Resolve -xml-annotations-description Errors

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 17-19

