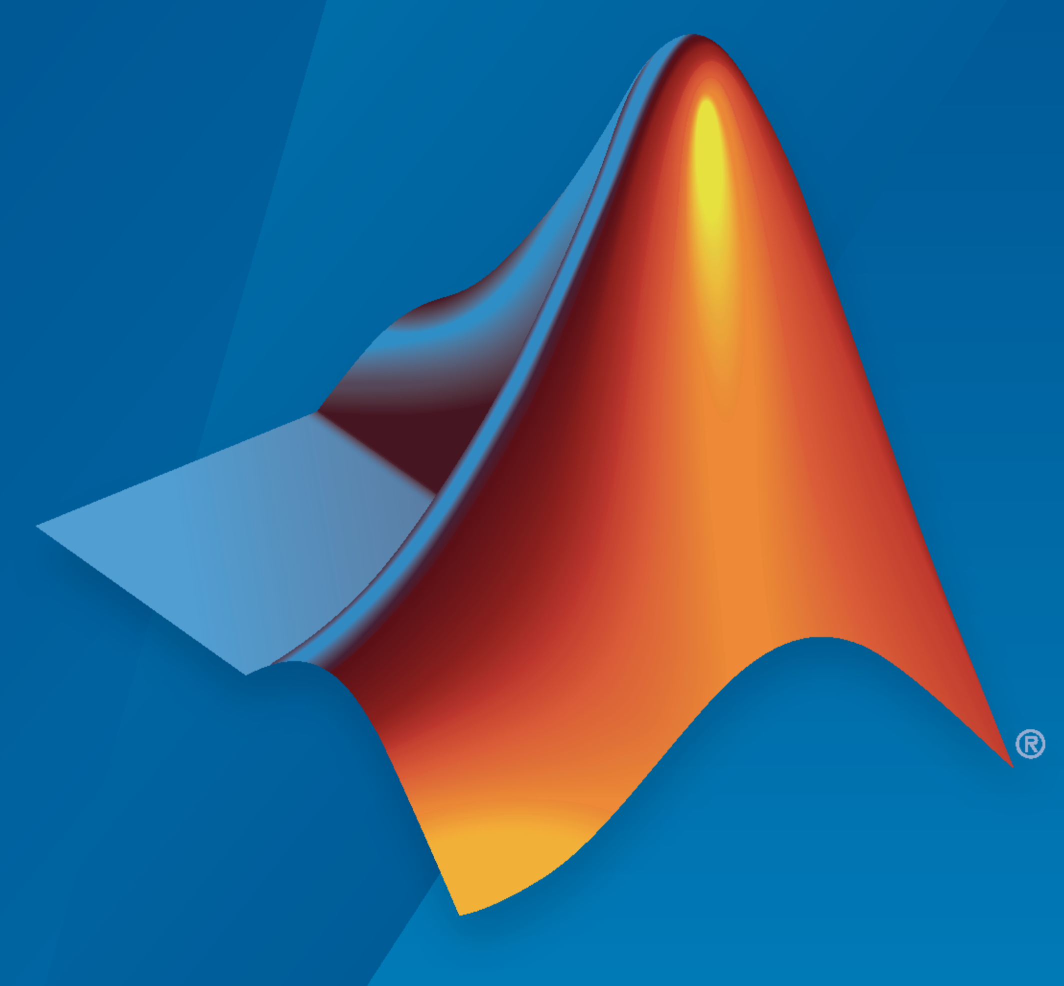
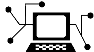
Polyspace® Code Prover™

User's Guide



R2023a

How to Contact MathWorks



Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Polyspace® Code Prover™ User's Guide

© COPYRIGHT 2013–2023 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

September 2013	Online Only	Revised for Version 9.0 (Release 2013b)
March 2014	Online Only	Revised for Version 9.1 (Release 2014a)
October 2014	Online Only	Revised for Version 9.2 (Release 2014b)
March 2015	Online Only	Revised for Version 9.3 (Release 2015a)
September 2015	Online Only	Revised for Version 9.4 (Release 2015b)
March 2016	Online Only	Revised for Version 9.5 (Release 2016a)
September 2016	Online Only	Revised for Version 9.6 (Release 2016b)
March 2017	Online Only	Revised for Version 9.7 (Release 2017a)
September 2017	Online Only	Revised for Version 9.8 (Release 2017b)
March 2018	Online Only	Revised for Version 9.9 (Release 2018a)
September 2018	Online Only	Revised for Version 9.10 (Release 2018b)
March 2019	Online Only	Revised for Version 10.0 (Release 2019a)
September 2019	Online Only	Revised for Version 10.1 (Release 2019b)
March 2020	Online Only	Revised for Version 10.2 (Release 2020a)
September 2020	Online Only	Revised for Version 10.3 (Release 2020b)
March 2021	Online Only	Revised for Version 10.4 (Release 2021a)
September 2021	Online Only	Revised for Polyspace Code Prover Version 10.5, Polyspace Code Prover Server Version 10.5, and Polyspace Code Prover Access Version 2.5 (Release 2021b)
March 2022	Online Only	Revised for Polyspace Code Prover Version 10.6, Polyspace Code Prover Server Version 10.6, and Polyspace Access Version 4.0 (Release 2022a)
September 2022	Online Only	Revised for Polyspace Code Prover Version 10.7, Polyspace Code Prover Server Version 10.7, and Polyspace Access Version 4.1 (Release 2022b)
March 2023	Online Only	Revised for Polyspace Code Prover Version 10.8, Polyspace Code Prover Server Version 10.8, and Polyspace Access Version 4.2 (Release 2023a)

Contents

1 Introduction

About This User's Guide 1-2

Configure Analysis on Desktop

2 Set Up Polyspace Projects on Desktop

Add Source Files for Analysis in Polyspace Desktop User Interface **2-2**
Polyspace Project and Source File Paths **2-2**
Add Sources from Build Command **2-3**
Add Sources Manually **2-5**
Add Source Files Based on AUTOSAR Design Specifications **2-6**

Contents of Polyspace Project and Results Folders **2-7**
File Organization **2-7**
Files in the Results Folder **2-7**

Create Polyspace Projects from Visual Studio Build **2-9**
Create Polyspace Project from Build in Visual Studio Developer Command Prompt **2-9**
Create Polyspace Project from Build in Visual Studio IDE **2-10**

Create Project in Polyspace Desktop User Interface Using Configuration Template **2-13**
Why Use Templates **2-13**
Use Predefined Template **2-13**
Create Your Own Template **2-13**
Sharing Project Templates **2-15**

Update Project in Polyspace Desktop User Interface **2-17**
Change Folder Path **2-17**
Refresh Source List **2-18**
Refresh Project Created from Build Command **2-18**
Add Source and Include Folders **2-18**
Manage Include File Sequence **2-18**

Modularize Large Projects in Polyspace Desktop User Interface . . **2-20**

Organize Layout of Polyspace Desktop User Interface **2-23**
Create Your Own Layout **2-23**
Save and Reset Layout **2-24**

Customize Polyspace Desktop User Interface **2-25**
Possible Customizations **2-25**
Storage of Polyspace User Interface Customizations **2-27**

Upload Results to Polyspace Access **2-28**
Upload Results from Polyspace Desktop Client **2-28**
Upload Results at Command Line **2-29**
Results Upload Compatibility and Permissions **2-29**

3 Run Polyspace Analysis on Desktop

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

Storage of Temporary Files During Polyspace Analysis **3-6**

4 Run Polyspace Analysis with Windows or Linux Scripts

Run Polyspace Analysis from Command Line **4-2**
Specify Sources and Analysis Options Directly **4-2**
Specify Sources and Analysis Options in Text File **4-2**
Create Options File from Build System **4-3**

Modularize Polyspace Analysis by Using Build Command **4-5**
Build Source Code **4-5**
Create One Polyspace Options File for Full Build **4-7**
Create Options File for Specific Binary in Build Command **4-8**
Create One Options File Per Binary Created in Build Command **4-8**

Select Files for Polyspace Analysis Using Pattern Matching **4-11**
When to Specify File Selection Patterns **4-11**
Supported Patterns for File Selection **4-12**

Modularize Polyspace Analysis at Command Line Based on an Initial Interdependency Analysis **4-15**
Basic Options **4-15**
Constrain Module Complexity During Partitioning **4-16**
Result Folder Names **4-16**
Forbid Cycles in Module Dependence Graph **4-16**

Configure Polyspace Analysis Options in User Interface and Generate Scripts . . . 4-17
Prerequisites . . . 4-18
Generate Scripts from Configuration . . . 4-18
Run Analysis with Generated Scripts . . . 4-19

5 Run Polyspace Analysis with MATLAB Scripts

Integrate Polyspace with MATLAB and Simulink . . . 5-2
Same Release of Polyspace and MATLAB . . . 5-2
MATLAB Release Earlier Than Polyspace . . . 5-3
Check Integration Between MATLAB and Polyspace . . . 5-4

Get Started with Polyspace Analysis by Using MATLAB . . . 5-5
Prerequisites . . . 5-5
Run Polyspace Analysis by Using MATLAB . . . 5-5
Frequently Used MATLAB Functions . . . 5-6

Run Polyspace Analysis by Using MATLAB Scripts . . . 5-9
Prerequisites . . . 5-9
Specify Multiple Source Files . . . 5-9
Check for MISRA C:2012 Violations . . . 5-10
Check for Specific Defects or Coding Rule Violations . . . 5-10
Find Files That Do Not Compile . . . 5-11
Run Analysis on Server . . . 5-11

Compare Results from Different Polyspace Runs by Using MATLAB Scripts . . . 5-13
Review Only New Results Compared to Last Run . . . 5-13
Review New Results and Unreviewed Results from Last Run . . . 5-14

Generate MATLAB Scripts from Polyspace User Interface . . . 5-16
Prerequisites . . . 5-16
Create Scripts from Polyspace Projects . . . 5-16

Troubleshoot Polyspace Analysis from MATLAB . . . 5-18
Prerequisites . . . 5-18
Capture Polyspace Analysis Errors in Error Log . . . 5-18

6 Run Polyspace Analysis in Simulink

Run Polyspace Analysis on Code Generated with Embedded Coder . . . 6-2
Prerequisites . . . 6-2
Generate and Analyze Code . . . 6-2
Review Analysis Results . . . 6-4
Annotate Blocks to Justify Issues . . . 6-5

Address Polyspace Results by Annotating Simulink Blocks **6-6**
Annotate Blocks Through Polyspace User Interface **6-6**
Annotate Blocks in Simulink Editor **6-8**

Changes in Polyspace Analysis Workflows in Simulink in R2019b . . **6-9**
Code Verification Workflow in a Nutshell **6-9**
Locate Pre-R2019b Menu Items in Simulink Toolstrip **6-9**

Run Polyspace on Code Generated by Using Previous Releases of Simulink **6-12**
Prerequisite **6-12**
Run a Cross-Release Polyspace Analysis **6-12**
Review Results **6-13**

Run Polyspace Analysis on Code Generated from Simulink Model **6-15**
Prerequisites **6-15**
Open Simulink Model for Polyspace Analysis **6-15**
Check for Run-Time Errors in Generated Code **6-16**
Review Analysis Results **6-17**
Trace and Fix Issues in the Model **6-17**
Check for Coding Rule Violations **6-22**
Annotate Blocks to Justify Results **6-23**

Fix Model Design Issues Found as Run-time Errors and Coding Rule Violations in Generated Code **6-25**
Prerequisites **6-25**
Open Model **6-25**
Detect and Fix Run-Time Errors **6-25**
Detect and Fix Coding Rule Violations **6-27**

Run Polyspace Analysis on Generated Code by Using Packaged Options Files **6-29**
Generate and Package Polyspace Options Files **6-29**
Run Polyspace Analysis by Using the Packaged Options Files **6-31**

Run Polyspace Analysis on Custom Code in Simulink Models **6-32**
Prerequisite **6-32**
Analyze Custom Code **6-32**
Review Analysis Results **6-33**

Run Polyspace Analysis on S-Function Code **6-35**
Prerequisites **6-35**
S-Function Analysis Workflow **6-35**
Compile S-Functions to Be Compatible with Polyspace **6-35**
Example S-Function Analysis **6-35**

Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts **6-37**
Prerequisites **6-37**
C/C++ Function Called Once in Model **6-37**
C/C++ Function Called Multiple Times in Model **6-40**

Run Polyspace Analysis on Custom Code in C Function Block **6-45**
Prerequisites **6-45**

Open Model for Running Polyspace Analysis on Custom Code in C Function Block **6-45**
Run Polyspace Analysis **6-46**
Identify Issues in C Code **6-47**
Fix Identified Issues **6-49**

Recommended Model Configuration Parameters for Polyspace Analysis **6-51**

Configure Polyspace Options in Simulink **6-53**
Configure Options **6-53**
Share and Reuse Configuration **6-55**

How Polyspace Analysis of Generated Code Works **6-56**

Default Polyspace Options for Code Generated with Embedded Coder **6-57**
Default Options **6-57**
Constraint Specification **6-57**
Recommended Polyspace options for Verifying Generated Code . . . **6-58**
Hardware Mapping Between Simulink and Polyspace **6-58**

External Constraints on Polyspace Analysis of Generated Code . . . **6-59**
Extract External Constraints from Model **6-59**
Storage Classes Supported for Constraint Extraction From Simulink Model **6-60**
Specify Custom External Constraints **6-60**

Run Polyspace Analysis on Code Generated with TargetLink **6-62**
Configure and Run Analysis **6-62**
Review Analysis Results **6-63**

Default Polyspace Options for Code Generated with TargetLink . . . **6-64**
TargetLink Support **6-64**
Default Options **6-64**
Lookup Tables **6-64**
Data Range Specification **6-65**
Code Generation Options **6-65**

Troubleshoot Navigation from Code to Model **6-66**
Links from Code to Model Do Not Appear **6-66**
Links from Code to Model Do Not Work **6-66**
Your Model Already Uses Highlighting **6-67**

Polyspace Support of MATLAB and Simulink from Different Releases **6-68**
Complete Integration **6-70**
Cross-Release Integration **6-70**
Partial Integration **6-71**
Navigate Back to Model **6-71**

7 Run Polyspace Analysis in MATLAB Coder

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

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

8 Running Polyspace on AUTOSAR Code

Using Polyspace in AUTOSAR Software Development **8-2**
Check if Implementation of Software Components Follow Specifications **8-2**
Assess Impact of Edits to Specifications **8-3**
Check Code Implementation for Run-time Errors and Mismatch with Specifications **8-3**
Check Code Implementation Against Specification Updates **8-4**

Choose Between Component-Based and Integration Analysis of AUTOSAR Code with Polyspace **8-5**
Definitions **8-5**
Similarities and Differences **8-5**
Choosing Between Component-Based and Integration Analysis **8-6**

Benefits of Polyspace for AUTOSAR **8-7**
Polyspace Modularizes Analysis Based on AUTOSAR Components . . . **8-8**
Polyspace Detects Mismatch Between Code and AUTOSAR XML Spec **8-11**

Run Polyspace on AUTOSAR Code **8-14**
Run Polyspace in User Interface **8-14**
Run Polyspace Using Scripts **8-17**

Select AUTOSAR XML (ARXML) and Code Files for Polyspace Analysis **8-19**
Adapt Linux find Command to Select Files **8-19**
File Selection Options **8-19**
File Selection Examples **8-20**
Root Folder Specification **8-21**

Troubleshoot Polyspace Analysis of AUTOSAR Code **8-22**
View Project Completion Status **8-22**
View Errors in AUTOSAR XML Parsing **8-23**
View Compilation Errors in Code **8-23**

Interpret Errors and Warnings in Polyspace Analysis of AUTOSAR Code **8-26**
Overview of File Structure **8-26**
See File Selections **8-27**
Run Analysis **8-27**
Interpret Warnings **8-27**

Run Polyspace on AUTOSAR Code with Conservative Assumptions **8-29**

Run Polyspace on AUTOSAR Code Using Build Command **8-31**
Run Code Prover Without Compilation Options **8-31**
Run Code Prover with Compilation Options from Build Command . . **8-32**

9 Run Polyspace Analysis in IDE Plugins

Run Polyspace Analysis on Eclipse Projects **9-2**
Configure and Run Analysis **9-3**
Review Analysis Results **9-5**

Specify Polyspace Compiler Options Through Eclipse Project **9-7**
Eclipse Refers Directly to Your Compilation Toolchain **9-7**
Eclipse Uses Your Compilation Toolchain Through Build Command **9-8**

Configure Analysis on Server

10 Run Polyspace Analysis on Servers

Run Polyspace Code Prover on Server and Upload Results to Web Interface **10-2**
Prerequisites **10-2**
Check Polyspace Installation **10-3**
Run Code Prover on Sample Files **10-3**
Sample Scripts for Code Prover Analysis on Servers **10-5**
Specify Sources and Options in Separate Files from Launching Scripts **10-5**
Complete Workflow **10-6**

Send Email Notifications with Polyspace Code Prover Server Results **10-8**
Creating E-mail Notifications **10-8**
Prerequisites **10-9**
Export Results for E-mail Attachments **10-10**

Assign Owners and Export Assigned Results 10-10

Offload Polyspace Analysis from Continuous Integration Server to Another Server **10-12**
Install Products **10-12**
Configure and Start Job Scheduler Services on Head Node and Worker Node **10-14**
Offload Analysis from Client Node **10-15**

Sample Scripts for Polyspace Analysis with Jenkins **10-17**
Extending Sample Scripts to Your Development Process **10-17**
Prerequisites **10-18**
Set Up Polyspace Plugin in Jenkins **10-19**
Script to Run Bug Finder, Upload Results and Send Common Notification **10-22**
Script to Run Bug Finder, Upload Results and Send Personalized Notification **10-24**

Sample Jenkins Pipeline Scripts for Polyspace Analysis **10-31**
Prerequisites **10-31**
Run Polyspace Analysis in Stages in a Pipeline Script **10-31**

Integrate Polyspace Server Products with MATLAB **10-33**
Integrate Polyspace Server Products with MATLAB **10-33**
Check Integration Between MATLAB and Polyspace **10-33**
Run Polyspace Server Products with MATLAB Scripts **10-34**

Configure Job Submissions from Desktop to Server

11 Offload Polyspace Analysis to Remote Servers from Desktop

Send Polyspace Analysis from Desktop to Remote Servers **11-2**
Client-Server Workflow for Running Analysis **11-2**
Prerequisites **11-3**
Offload Analysis in Polyspace User Interface **11-3**

Send Polyspace Analysis from Desktop to Remote Servers Using Scripts **11-5**
Client-Server Workflow for Running Analysis **11-5**
Prerequisites **11-6**
Run Remote Analysis **11-6**
Manage Remote Analysis **11-7**
Sample Scripts for Remote Analysis **11-9**

12 Configure Polyspace Analysis

Specify Polyspace Analysis Options **12-2**
Polyspace User Interface **12-2**
Windows or Linux Scripts **12-2**
MATLAB Scripts **12-3**
Eclipse and Eclipse-based IDEs **12-3**
Simulink **12-3**
MATLAB Coder App **12-3**

Options Files for Polyspace Analysis **12-5**
What are Options Files **12-5**
Specifying Options Files **12-5**
Specifying Multiple Options Files **12-6**

13 Configure Target and Compiler Options

Specify Target Environment and Compiler Behavior **13-2**
Extract Options from Build Command **13-2**
Specify Options Explicitly **13-3**

C/C++ Language Standard Used in Polyspace Analysis **13-5**
Supported Language Standards **13-5**
Default Language Standard **13-5**

C11 Language Elements Supported in Polyspace **13-8**

C++11 Language Elements Supported in Polyspace **13-9**

C++14 Language Elements Supported in Polyspace **13-12**

C++17 Language Elements Supported in Polyspace **13-15**

Provide Standard Library Headers for Polyspace Analysis **13-19**

Create Polyspace Analysis Configuration from Build Command (Makefile) **13-21**

Requirements for Project Creation from Build Systems **13-23**
Compiler Requirements **13-23**
Build Command Requirements **13-24**

Supported Keil or IAR Language Extensions **13-26**
Special Function Register Data Type **13-26**

Keywords Removed During Preprocessing **13-27**

Remove or Replace Keywords Before Compilation **13-28**
Remove Unrecognized Keywords **13-28**
Remove Unrecognized Function Attributes **13-30**

Gather Compilation Options Efficiently **13-31**

14 Configure Inputs and Stubbing Options

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

External Constraints for Polyspace Analysis **14-6**
Effect of External Constraints **14-6**
Constraint Specification **14-7**
Constraint Specification Limitations **14-11**

Constrain Global Variable Range for Polyspace Analysis **14-12**
User Interface (Desktop Products Only) **14-12**
Command Line **14-13**

Constrain Function Inputs for Polyspace Analysis **14-14**
User Interface (Desktop Products Only) **14-14**
Command Line **14-15**

XML File Format for Polyspace Analysis Constraints **14-17**
Syntax Description — XML Elements **14-17**
Valid Modes and Default Values **14-21**

15 Configure Multitasking Analysis

Analyze Multitasking Programs in Polyspace **15-2**
Configure Analysis **15-3**
Review Analysis Results **15-4**
Differences Between Bug Finder and Code Prover **15-5**

Auto-Detection of Thread Creation and Critical Section in Polyspace **15-7**
Multitasking Routines that Polyspace Can Detect **15-7**
Example of Automatic Thread Detection **15-8**
Naming Convention for Automatically Detected Threads **15-11**
Limitations of Automatic Thread Detection **15-12**

Configuring Polyspace Multitasking Analysis Manually 15-17
Specify Options for Multitasking Analysis 15-17
Adapt Code for Code Prover Multitasking Analysis 15-17

Protections for Shared Variables in Multitasking Code 15-21
Detect Unprotected Access 15-21
Protect Using Critical Sections 15-22
Protect Using Temporally Exclusive Tasks 15-23
Protect Using Priorities 15-23
Protect By Disabling Interrupts 15-24

Define Atomic Operations in Multitasking Code 15-25
Nonatomic Operations 15-25
What Polyspace Considers as Nonatomic 15-25
Define Specific Operations as Atomic 15-26

Define Task Priorities for Data Race Detection in Polyspace 15-28
Emulating Task Priorities 15-28
Examples of Task Priorities 15-28
Further Explorations 15-29
Effect of Task Priorities in Code Prover 15-29

Define Critical Sections with Functions That Take Arguments . . . 15-31
Polyspace Assumption on Functions Defining Critical Sections . . . 15-31
Adapt Polyspace Analysis to Lock and Unlock Functions with Arguments 15-31

16 Configure Coding Rules Checking and Code Metrics Computation

Check for and Review Coding Standard Violations 16-2
Configure Coding Rules Checking 16-2
Review Coding Rule Violations 16-6
Generate Reports 16-6

Avoid Violations of MISRA C:2012 Rules 8.x 16-8

Software Quality Objective Subsets (C:2004) 16-11
Rules in SQO-Subset1 16-11
Rules in SQO-Subset2 16-12

Software Quality Objective Subsets (AC AGC) 16-15
Rules in SQO-Subset1 16-15
Rules in SQO-Subset2 16-16

Software Quality Objective Subsets (C:2012) 16-19
Guidelines in SQO-Subset1 16-19
Guidelines in SQO-Subset2 16-20

Software Quality Objective Subsets (C++) 16-23
SQO Subset 1 – Direct Impact on Selectivity 16-23

SQO Subset 2 – Indirect Impact on Selectivity **16-24**

Coding Rule Subsets Checked Early in Analysis **16-29**
MISRA C:2004 and MISRA AC AGC Rules **16-29**
MISRA C:2012 Rules **16-36**

Create Custom Coding Rules **16-44**
Specify Naming Convention **16-44**
Check for Violations of Defined Custom Coding Rule **16-46**

Compute Code Complexity Metrics Using Polyspace **16-47**
Impose Limits on Metrics (Desktop Products Only) **16-47**
Impose Limits on Metrics (Server and Access products) **16-49**

HIS Code Complexity Metrics **16-50**
Project **16-50**
File **16-50**
Function **16-50**

Migrate Code Prover Workflows for Checking Coding Standards and Code Metrics to Bug Finder **16-52**
Changes in Workflow **16-52**

17 Polyspace Coverage of Coding Standards

Polyspace Support for Coding Standards **17-2**
Summary of Polyspace Support **17-2**
AUTOSAR C++14 . . . . **17-2**
MISRA C++:2008 . . . . **17-3**
MISRA C:2012 . . . . **17-4**
CERT C **17-7**
Other **17-8**

MISRA C:2004 and MISRA AC AGC Coding Rules **17-9**
Supported MISRA C:2004 and MISRA AC AGC Rules **17-9**
Troubleshooting **17-9**
List of Supported Coding Rules **17-9**
Unsupported MISRA C:2004 and MISRA AC AGC Rules **17-41**

Required or Mandatory MISRA C:2012 Rules Supported by Polyspace Bug Finder **17-43**
Mandatory Rules **17-43**
Required Rules **17-44**

Decidable MISRA C:2012 Rules Supported by Polyspace Bug Finder **17-54**

Undecidable MISRA C:2012 Rules and Directives Supported by Polyspace Bug Finder **17-64**
Undecidable Rules **17-64**
Undecidable Directives **17-67**

Essential Types in MISRA C:2012 Rules 10.x **17-69**
Categories of Essential Types **17-69**
How MISRA C:2012 Uses Essential Types **17-69**

Unsupported MISRA C:2012 Guidelines **17-71**

Required and Statically Enforceable CERT C Rules Supported by Polyspace Bug Finder **17-72**

Required MISRA C++:2008 Coding Rules Supported by Polyspace Bug Finder **17-80**
Supported Rules **17-80**
Unsupported Rules **17-96**

JSF AV C++ Coding Rules **17-97**
Supported JSF C++ Coding Rules **17-97**
Unsupported JSF++ Rules **17-115**

Required AUTOSAR C++14 Coding Rules Supported by Polyspace Bug Finder **17-122**
Supported Rules **17-122**
Unsupported Rules **17-150**

Statically Enforceable AUTOSAR C++14 Rules Supported by Polyspace Bug Finder **17-153**
Automated Rules **17-153**
Partially Automated Rules **17-179**

18 Configure Verification of Modules or Libraries

Provide Context for C Code Verification **18-2**
Control Variable Range **18-2**
Control Function Call Sequence **18-2**
Control Stubbing Behavior **18-2**

Provide Context for C++ Code Verification **18-4**
Control Variable Range **18-4**
Control Function Call Sequence **18-4**

Verify C Application Without main Function **18-6**
Generate main Function **18-6**
Manually Write main Function **18-6**

Verify C++ Classes **18-9**
Verification of Classes **18-9**
Methods and Class Specifics **18-11**

Constrain Variable Ranges for Polyspace Analysis Using Manual Stubs and Manual main() Function **18-18**
Code Prover Assumptions About Variable Ranges at Program Boundary **18-18**

Define Manual main() and Stubs to Constrain Ranges at Program Boundary **18-19**

19 Configure Code Prover Run-Time Checks

Modify or Disable Code Prover Run-Time Checks **19-2**
Integer Overflow **19-2**
Floating Point Overflow **19-3**
Initialization **19-3**
Library Functions **19-3**
Pointers **19-4**
Unreachable Code or Dead Code **19-4**

Modify Code Prover Run-Time Checks Through Code Behavior Specifications **19-5**
Mapping to Standard Function for Precision Improvement **19-5**
Mapping to Standard Function for Concurrency Detection **19-6**
Extending Initialization Checks **19-7**
Specifying Contribution from Stubbed Functions to Stack Size **19-7**

20 Configure Comment Import from Previous Results

Import Review Information from Previous Polyspace Analysis **20-2**
Automatic Import from Last Analysis **20-2**
Import from Another Analysis Result **20-2**
Import Algorithm **20-3**
View Imported Review Information That Does Not Apply **20-4**

Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results **20-5**
Mapping Multiple MISRA C: 2004 Annotations to the Same MISRA C: 2012 Result **20-6**

Review Results in Polyspace User Interface

21 Interpret Polyspace Code Prover Results

Interpret Code Prover Results in Polyspace Desktop User Interface **21-2**
Interpret Result **21-3**

Find Root Cause of Result **21-4**

Dashboard in Polyspace Desktop User Interface **21-7**
Code Covered by Verification **21-7**
Check Distribution **21-9**
Top 5 Orange Sources **21-9**
Top 5 Coding Rule Violations **21-10**
Other Dashboard Features **21-10**

Concurrency Modeling in Polyspace Desktop User Interface **21-11**

Results List in Polyspace Desktop User Interface **21-12**

Source Code in Polyspace Desktop User Interface **21-15**
Examine Source Code **21-15**
View Variable Ranges **21-17**
View Extent of Code Block **21-18**
Manage Multiple Files **21-18**
Expand and Collapse Macros **21-19**
See Function Callers and Callees **21-20**

Result Details in Polyspace Desktop User Interface **21-21**
View Traceback **21-21**
View Error Call Graph **21-22**
View Call Hierarchy and Variable Access **21-23**

Call Hierarchy in Polyspace Desktop User Interface **21-24**
Show or Hide Callers and Callees **21-25**
Navigate Call Hierarchy **21-25**
Determine if Function is Stubbed **21-25**
See Entire Call Hierarchy of Program **21-25**

Variable Access in Polyspace Desktop User Interface **21-27**
View Access Graph **21-29**
View Structured Variables **21-30**
View Operations on Anonymous Variables **21-31**
View Access Through Global Pointers **21-31**
Show or Hide Callers and Callees **21-32**
Show or Hide Accesses in Unreachable Code **21-32**
Other Features **21-32**

Understanding Changes in Polyspace Results After Product Upgrade **21-33**
Changes in Polyspace Code Prover Results **21-33**
Changes in Polyspace Bug Finder Results **21-35**

22 Reviewing Checks

Review and Fix Absolute Address Usage Checks **22-2**

Review and Fix Correctness Condition Checks **22-3**
Step 1: Interpret Check Information **22-3**
Step 2: Determine Root Cause of Check **22-5**
Step 3: Trace Check to Polyspace Assumption **22-6**

Review and Fix Division by Zero Checks **22-7**
Step 1: Interpret Check Information **22-7**
Step 2: Determine Root Cause of Check **22-8**
Step 3: Look for Common Causes of Check **22-9**
Step 4: Trace Check to Polyspace Assumption **22-10**

Review and Fix Function Not Called Checks **22-11**
Step 1: Interpret Check Information **22-11**
Step 2: Determine Root Cause of Check **22-11**
Step 3: Look for Common Causes of Check **22-12**

Review and Fix Function Not Reachable Checks **22-13**
Step 1: Interpret Check Information **22-13**
Step 2: Determine Root Cause of Check **22-13**

Review and Fix Function Not Returning Value Checks **22-15**
Step 1: Interpret Check Information **22-15**
Step 2: Determine Root Cause of Check **22-15**

Review and Fix Illegally Dereferenced Pointer Checks **22-17**
Step 1: Interpret Check Information **22-17**
Step 2: Determine Root Cause of Check **22-19**
Step 3: Look for Common Causes of Check **22-20**
Step 4: Trace Check to Polyspace Assumption **22-21**

Review and Fix Incorrect Object Oriented Programming Checks **22-22**
Step 1: Interpret Check Information **22-22**
Step 2: Determine Root Cause of Check **22-22**

Review and Fix Invalid C++ Specific Operations Checks **22-24**
Step 1: Interpret Check Information **22-24**
Step 2: Determine Root Cause of Check **22-25**
Step 3: Trace Check to Polyspace Assumption **22-25**

Review and Fix Invalid Shift Operations Checks **22-26**
Step 1: Interpret Check Information **22-26**
Step 2: Determine Root Cause of Check **22-27**
Step 3: Look for Common Causes of Check **22-29**
Step 4: Trace Check to Polyspace Assumption **22-29**

Review and Fix Invalid Use of Standard Library Routine Checks . **22-30**
Step 1: Interpret Check Information **22-30**
Step 2: Trace Check to Polyspace Assumption **22-31**

Invalid Use of Standard Library Floating Point Routines **22-32**
What the Check Looks For **22-32**
Single-Argument Functions Checked **22-33**
Functions with Multiple Arguments **22-33**

Review and Fix Non-initialized Local Variable Checks **22-35**
Step 1: Interpret Check Information **22-35**
Step 2: Determine Root Cause of Check **22-35**
Step 3: Look for Common Causes of Check **22-36**
Step 4: Trace Check to Polyspace Assumption **22-37**

Review and Fix Non-initialized Pointer Checks **22-38**
Step 1: Interpret Check Information **22-38**
Step 2: Determine Root Cause of Check **22-38**
Step 3: Trace Check to Polyspace Assumption **22-39**

Review and Fix Non-initialized Variable Checks **22-40**
Step 1: Interpret Check Information **22-40**
Step 2: Determine Root Cause of Check **22-40**
Step 3: Trace Check to Polyspace Assumption **22-41**

Review and Fix Non-Terminating Call Checks **22-42**
Step 1: Determine Root Cause of Check **22-42**
Step 2: Look for Common Causes of Check **22-43**

Identify Function Call with Run-Time Error **22-44**

Review and Fix Non-Terminating Loop Checks **22-46**
Step 1: Interpret Check Information **22-46**
Step 2: Determine Root Cause of Check **22-46**
Step 3: Look for Common Causes of Check **22-47**

Identify Loop Operation with Run-Time Error **22-49**

Review and Fix Null This-pointer Calling Method Checks **22-51**
Step 1: Interpret Check Information **22-51**
Step 2: Determine Root Cause of Check **22-51**

Review and Fix Out of Bounds Array Index Checks **22-53**
Step 1: Interpret Check Information **22-53**
Step 2: Determine Root Cause of Check **22-54**
Step 3: Look for Common Causes of Check **22-55**
Step 4: Trace Check to Polyspace Assumption **22-56**

Review and Fix Overflow Checks **22-57**
Step 1: Interpret Check Information **22-57**
Step 2: Determine Root Cause of Check **22-58**
Step 3: Look for Common Causes of Check **22-59**
Step 4: Trace Check to Polyspace Assumption **22-60**

Detect Overflows in Buffer Size Computation **22-61**

Review and Fix Return Value Not Initialized Checks **22-63**
Step 1: Interpret Check Information **22-63**
Step 2: Determine Root Cause of Check **22-63**
Step 3: Look for Common Causes of Check **22-64**
Step 4: Trace Check to Polyspace Assumption **22-65**

Review and Fix Uncaught Exception Checks **22-66**
Step 1: Interpret Check Information **22-66**

Step 2: Determine Root Cause of Check **22-66**

Review and Fix Unreachable Code Checks **22-68**
Step 1: Interpret Check Information **22-68**
Step 2: Determine Root Cause of Check **22-68**
Step 3: Look for Common Causes of Check **22-70**

Review and Fix User Assertion Checks **22-73**
Step 1: Determine Root Cause of Check **22-73**
Step 2: Look for Common Causes of Check **22-75**
Step 3: Trace Check to Polyspace Assumption **22-75**

Find Relations Between Variables in Code **22-77**
Insert Pragma to Determine Variable Relation **22-77**
Further Exploration **22-79**

Review Polyspace Results on AUTOSAR Code **22-80**
Open Results **22-80**
See Overview of Results for all Software Components **22-81**
See Runnables and Source Files in Software Component **22-82**
Interpret AUTOSAR Specific Run-time Checks for Software Component **22-85**

23 Fix or Comment Polyspace Results

Address Results in Polyspace User Interface Through Bug Fixes or Justifications **23-2**
Add Review Information to Results File **23-2**
Comment or Annotate in Code **23-3**

24 Manage Results

Filter and Group Results in Polyspace Desktop User Interface **24-2**
Filter Results **24-3**
Group Results **24-7**

Prioritize Check Review **24-9**

25 Generate Reports from Polyspace Results

Generate Reports from Polyspace Results **25-2**
Generate Reports from User Interface **25-2**
Generate Reports from Command Line **25-3**

Export Polyspace Analysis Results **25-5**
Export Results to Text File **25-5**
Export Results to MATLAB Table **25-5**
Export Results to JSON Format **25-6**
View Exported Results **25-6**

Export Polyspace Analysis Results to Excel by Using MATLAB Scripts **25-9**
Report Result Summary and Details in One Worksheet **25-9**
Control Formatting of Excel Report **25-10**

Export Global Variable List **25-11**
Export Variable List to Text File **25-11**
Export Variable List to MATLAB Table **25-12**
View Exported Variable List **25-12**

Visualize Code Prover Analysis Results in MATLAB **25-14**
Export Results to MATLAB Table **25-14**
Generate Graphs from Results and Include in Report **25-14**

Customize Existing Code Prover Report Template **25-17**
Prerequisites **25-17**
View Components of Template **25-17**
Change Components of Template **25-18**
Further Exploration **25-21**

Sample Report Template Customizations **25-22**
Add List of Recursive Functions **25-22**
Show Red Run-Time Checks Only **25-22**
Show Non-Justified Run-Time Checks Only **25-23**
Add Chapter for Functional Design Errors **25-24**

Generate Report Containing MISRA C:2012 Violations, Code Metrics, and Runtime Check Results **25-25**
Prerequisite **25-25**
Obtain Code Metrics and Coding Rules Results by Using Bug Finder **25-25**
Obtain Run Time Check and Stack Usage Results by Using Code Prover **25-26**
Generate a Combined Report **25-27**

Review Results on Web Browser

26 Interpret Polyspace Code Prover Access Results

Interpret Code Prover Results in Polyspace Access Web Interface **26-2**
Interpret Result **26-3**
Find Root Cause of Result **26-4**

Dashboard in Polyspace Access Web Interface **26-7**

Code Metrics Dashboard in Polyspace Access Web Interface **26-9**

Quality Objectives Dashboard in Polyspace Access **26-12**
Monitor Code Quality Against Software Quality Objectives **26-12**
Customize Software Quality Objectives **26-14**

Results List in Polyspace Access Web Interface **26-17**

Source Code in Polyspace Access Web Interface **26-19**
Tooltips **26-19**
Examine Source Code **26-20**
Expand Macros **26-21**
View Code Block **26-22**
Navigate from Code to Model **26-22**

Result Details in Polyspace Access Web Interface **26-24**

Call Hierarchy in Polyspace Access Web Interface **26-26**

Configuration Settings in Polyspace Access Web Interface **26-28**

Global Variables in Polyspace Access Web Interface **26-31**

Review History in Polyspace Access Web Interface **26-36**

Create Bug Tracking Tool Tickets from the Polyspace Access Web Interface **26-38**
Create a Ticket **26-38**
Manage Existing Tickets **26-39**

27 Fix or Comment Polyspace Results on Web Browser

Address Results in Polyspace Access Through Bug Fixes or Justifications **27-2**
Add Review Information in Result Details pane **27-2**
Comment or Annotate in Code **27-4**

Import Review Information from Existing Polyspace Access Projects **27-5**
Import Review Information from Source Project to Target Project in Polyspace Access **27-5**
View and Select Imported Reviews **27-6**
Confirm Imported Review Information **27-6**
Import Review Information at the Command-Line **27-7**

28 Manage Results

Manage Permissions and View Project Trends in Polyspace Access Web Interface **28-2**
Create a Project Folder **28-2**
Manage Project Permissions **28-3**
View Project Trends **28-6**

Filter and Sort Results in Polyspace Access Web Interface **28-8**
Filter Results **28-10**

Create Custom Filter Groups in Polyspace Access Web Interface **28-13**

Manage Software Quality Objectives in Polyspace Access **28-15**
Manage SQOs in the User Interface **28-15**
Manage SQOs at the Command Line **28-16**

Add Labels to Project Runs in Polyspace Access **28-18**
Manage Labels in the User Interface **28-18**
Manage Labels at the Command Line **28-19**

Prioritize Check Review in Polyspace Access Web Interface **28-21**

Compare Results in Polyspace Access Project to Previous Runs and View Trends **28-23**
Comparison Mode in the Polyspace Access Interface **28-23**
Comparison Mode at the Command Line **28-25**

29 Export Results from Polyspace Access Web Server

Open or Export Results from Polyspace Access **29-2**
Open Polyspace Access Results in a Desktop Interface **29-2**
Export Polyspace Access Results to a TSV File **29-2**

Generate Report and Variables List from Polyspace Access **29-4**

30 Hide Known or Acceptable Results Using Code Annotations

Annotate Code and Hide Known or Acceptable Results **30-2**
Code Annotation Syntax **30-2**
Syntax Examples **30-7**
Code Annotation Warnings **30-10**
Ignoring Code Annotations **30-10**

Short Names of Code Prover Run-Time Checks **30-12**

Short Names of Code Complexity Metrics **30-14**
Project Metrics **30-14**
File Metrics **30-14**
Function Metrics **30-14**

Annotate Code for Known or Acceptable Results (Not Recommended) **30-16**
Add Annotations from the Polyspace Interface **30-16**
Add Annotations Manually **30-17**

Define Custom Annotation Format **30-20**
Define Annotation Syntax Format **30-22**
Map Your Annotation to the Polyspace Annotation Syntax **30-25**
Define Multiple Custom Annotation Syntaxes **30-26**

Annotation Description Full XML Template **30-28**
Example **30-31**

31 Advanced Review Workflows

Evaluate Polyspace Code Prover Results Against Software Quality Objectives **31-2**
Specifications of SQO Levels **31-2**
Compare Verification Results Against Software Quality Objectives **31-7**
Customize SQO Levels **31-8**

Justify Coding Rule Violations Using Code Prover Checks **31-9**
Rules About Data Type Conversions **31-9**
Rules About Pointer Arithmetic **31-11**

Polyspace Results in Lines Containing Macros **31-14**
Macros in Source Lines Can Be Expanded in Place **31-14**

Results in Function-Like Macros Shown Only Once 31-14

Migrate Results from Polyspace Metrics to Polyspace Access . . . 31-16
Requirements for Migration 31-17
Migration of Results 31-18
Differences in SQO Between Polyspace Metrics and Polyspace Access 31-19

Understanding Code Prover Results

32 Code Prover Checks and Source Code Tooltips

Code Prover Result and Source Code Colors 32-2
Result Colors 32-2
Source Code Colors 32-4
Global Variable Colors 32-5

Reviewing Code Prover Run-Time Checks 32-7
Data Flow Checks 32-7
Numerical Checks 32-7
Static Memory Checks 32-8
Control Flow Checks 32-8
C++ Checks 32-8
Other Checks 32-9

Code Prover Analysis Following Red and Orange Checks 32-10
Code Following Red Check 32-10
Green Check Following Orange Check 32-11
Gray Check Following Orange Check 32-11
Red Check Following Orange Check 32-12
Red and Green Checks in Unreachable Code 32-12

Order of Code Prover Run-Time Checks 32-15

Variable Ranges in Source Code Tooltips After Code Prover Analysis 32-17
Why Code Prover Reports Ranges on Variables 32-17
Why Variable Ranges Can Sometimes Be Narrower Than Expected 32-18

33 Orange Checks in Polyspace Code Prover

Orange Checks in Polyspace Code Prover 33-2
When Orange Checks Occur 33-2

Why Review Orange Checks 33-3
How to Review Orange Checks 33-3
How to Reduce Orange Checks 33-3

Managing Orange Checks in Polyspace Code Prover **33-5**
Software Development Stage 33-6
Quality Goals 33-7

Critical Orange Checks in Polyspace Code Prover **33-9**
How to See Only Critical Checks 33-9
Which Orange Checks Are Considered Critical 33-11

Limit Display of Orange Checks in Polyspace Desktop User Interface **33-14**

Reduce Orange Checks in Polyspace Code Prover **33-17**
Provide Context for Verification 33-17
Improve Verification Precision 33-18
Follow Coding Rules 33-18
Reduce Application Size 33-19
Follow Verification Setup Suggestions 33-19

Troubleshooting

34 Troubleshooting in Polyspace Code Prover

View Error Information When Analysis Stops **34-3**
View Error Information in User Interface 34-3
View Error Information in Log File 34-3

Troubleshoot Compilation and Linking Errors **34-6**
Issue 34-6
Possible Cause: Deviations from Standard 34-6
Possible Cause: Linking Errors 34-7
Possible Cause: Conflicts with Polyspace Function Stubs 34-8

Reduce Memory Usage and Time Taken by Polyspace Analysis **34-10**
Issue 34-10
Possible Cause: Temporary Folder on Network Drive 34-10
Possible Cause: Anti-Virus Software 34-10
Possible Cause: Large and Complex Application 34-11
Possible Cause: Too Many Entry Points for Multitasking Applications 34-13

Identify Root Causes of Code Prover Red or Orange Checks **34-15**
Issue 34-15
Possible Cause: Relation to Prior Code Operations 34-15
Possible Cause: Software Assumptions 34-16

Contact Technical Support About Issues with Running Polyspace 34-18
Provide System Information 34-18
Provide Information About the Issue 34-19
Provide Polyspace Analysis Statistics File (Optional) 34-20

Fix Error: Polyspace Cannot Find Server 34-21
Message 34-21
Possible Cause 34-21
Solution 34-21

Fix Error: Job Manager Cannot Write to Database 34-22
Message 34-22
Possible Cause 34-22
Workaround 34-22

Resolve Error: No Compilation Unit Detected in Your Build 34-23
Issue 34-23
Possible Solutions 34-23

Create Polyspace Projects from Build Systems That Use Unsupported Compilers 34-25
Issue 34-25
Cause 34-25
Solution 34-25

Fix Slow Build Process When Polyspace Traces Build 34-31
Issue 34-31
Cause 34-31
Solution 34-31

Check if Polyspace Supports Build Scripts 34-32
Issue 34-32
Possible Cause 34-32
Solution 34-32

Troubleshoot Project Creation from MinGW Build 34-33
Issue 34-33
Cause 34-33
Solution 34-33

Troubleshoot Project Creation from Visual Studio Build 34-34

Resolve polyspace-autosar Error: Could Not Find Include File 34-35
Issue 34-35
Possible Solutions 34-35

Resolve polyspace-autosar Error: Conflicting Universal Unique Identifiers (UUIDs) 34-37
Issue 34-37
Possible Solutions 34-37

Resolve polyspace-autosar Error: Data Type Not Recognized 34-38
Issue 34-38
Possible Solutions 34-38

Fix Polyspace Compilation Errors About Undefined Identifiers . . . **34-40**
Issue **34-40**
Possible Cause: Missing Files **34-40**
Possible Cause: Unrecognized Keyword **34-40**
Possible Cause: Declaration Embedded in #ifdef Statements **34-41**
Possible Cause: Project Created from Non-Debug Build **34-41**

Fix Polyspace Compilation Errors About Unknown Function Prototype **34-43**
Issue **34-43**
Cause **34-43**
Solution **34-43**

Fix Polyspace Compilation Errors Related to #error Directive . . . **34-44**
Issue **34-44**
Cause **34-44**
Solution **34-44**

Fix Polyspace Compilation Errors About Large Objects **34-45**
Issue **34-45**
Cause **34-45**
Solution **34-45**

Fix Polyspace Compilation Errors Related to Generic Compiler . . **34-47**
Issue **34-47**
Cause **34-47**
Solution **34-47**

Fix Polyspace Compilation Errors Related to GNU Compiler **34-48**
Issue **34-48**
Cause **34-48**
Solution **34-48**

Fix Polyspace Compilation Errors Related to Visual Compilers . . **34-49**
Import Folder **34-49**
pragma Pack **34-49**
C++/CLI **34-50**

Fix Polyspace Compilation Errors Related to Keil or IAR Compiler **34-51**
Missing Identifiers **34-51**

Fix Polyspace Compilation Errors Related to Diab Compiler **34-52**
Issue **34-52**
Cause **34-52**
Solution **34-52**

Fix Polyspace Compilation Errors Related to Green Hills Compiler **34-54**
Issue **34-54**
Cause **34-54**
Solution **34-54**

Fix Polyspace Compilation Errors Related to TASKING Compiler **34-56**
Issue **34-56**
Cause **34-56**
Solution **34-56**

Fix Polyspace Compilation Errors Related to Texas Instruments Compilers **34-58**
Issue **34-58**
Possible Solutions **34-58**

Fix Polyspace Compilation Errors About In-Class Initialization (C++) **34-59**

Fix Polyspace Linking Errors About Conflicting Declarations in Different Translation Units **34-60**
Issue **34-60**
Possible Cause: Variable Declaration and Definition Mismatch . . . **34-61**
Possible Cause: Function Declaration and Definition Mismatch . . . **34-61**
Possible Cause: Conflicts from Unrelated Declarations **34-62**
Possible Cause: Macro-dependent Definitions **34-63**
Possible Cause: Keyword Redefined as Macro **34-63**
Possible Cause: Differences in Structure Packing **34-64**

Fix Errors from Use of Polyspace Header Files **34-65**
Issue **34-65**
Possible Solutions **34-65**

Fix Polyspace Linking Errors Related to extern "C" Blocks **34-67**
Extern C Functions **34-67**
Functional Limitations on Some Stubbed Standard ANSI Functions **34-67**

Fix Polyspace Compilation Errors About Namespace std Without Prefix **34-69**
Issue **34-69**
Cause **34-69**
Solution **34-69**

Fix Polyspace Compilation Warnings Related to Assertion or Memory Allocation Functions **34-70**
Issue **34-70**
Cause **34-70**
Solution **34-70**

Troubleshoot Java Incompatibility in Polyspace Plugin for Eclipse **34-71**
Issue **34-71**
Possible Solutions **34-71**

Fix Issues When when Integrating Polyspace with MATLAB and Simulink **34-73**
Issue **34-73**
Possible Solutions **34-73**

Check Why Polyspace Functions are Unavailable in MATLAB **34-75**
Issue **34-75**
Possible Solution **34-75**

Fix MATLAB Crashes Referring to Polyspace in matlabroot **34-76**
Issue **34-76**
Possible Solutions **34-76**

Reasons for Unchecked Code **34-77**
Issue **34-77**
Possible Cause: Compilation Errors **34-78**
Possible Cause: Early Red or Gray Check **34-78**
Possible Cause: Incorrect Options **34-80**
Possible Cause: main Function Does Not Terminate **34-80**

Identify Why Some Files or Functions are Missing in Polyspace Results **34-83**
Issue **34-83**
Possible Cause: Files Not Verified **34-83**
Possible Cause: Filters Applied **34-84**

Fix Polyspace Overapproximations on Standard Library Math Functions **34-86**
Issue **34-86**
Cause **34-86**
Solution **34-86**

Avoid Red Checks in Unreachable Code When Using C++ STL Containers **34-87**
Issue **34-87**
Possible Solutions **34-87**

Fix Insufficient Memory Errors During Polyspace Report Generation **34-88**
Issue **34-88**
Possible Solutions **34-88**

Fix Polyspace Errors Related to Temporary Files **34-91**
No Access Rights **34-91**
No Space Left on Device **34-91**
Cannot Open Temporary File **34-91**

Fix Errors or Slow Polyspace Runs from Disk Defragmentation and Anti-virus Software **34-93**
Issue **34-93**
Possible Cause **34-93**
Solution **34-93**

Fix SQLite I/O Errors on Running Polyspace **34-95**
Issue **34-95**
Possible Solutions **34-95**

Fix License Error –4,0 When Running Polyspace **34-96**
Issue **34-96**
Possible Cause: Another Polyspace Instance Running **34-96**

Possible Cause: Prior Polyspace Run in Simulink or MATLAB Coder **34-96**

Fix Errors Applying Custom Annotation Format for Polyspace Results **34-97**
Issue **34-97**
Possible Solutions **34-97**

35 Troubleshooting Polyspace Access

Polyspace Access ETL and Web Server services do not start **35-2**
Issue **35-2**
Possible Cause: Hyper-V Network Configuration Cannot Resolve Local Host Names **35-2**

Contact Technical Support About Polyspace Access Issues **35-5**

1

Introduction

About This User's Guide

This User's Guide covers all Polyspace Code Prover products:

- Polyspace Code Prover™
- Polyspace Code Prover Server™
- Polyspace Access™

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

- **Desktop**: If you are running an analysis and reviewing the results on your desktop, you use Polyspace Code Prover. For desktop-specific workflows, see "Configure Analysis on Desktop" or "Review Results in Polyspace User Interface".
- **Server**: Running an analysis on a server, or reviewing the results from a server run on a web browser, you use:

  - Polyspace Code Prover Server to run the analysis.
  - Polyspace Access to host the analysis results (for review on a web browser).

  For server-specific workflows, see "Configure Analysis on Server" or "Review Results on Web Browser".

The Code Prover analysis engine underlies all Code Prover products. Chapters that do not mention a particular platform typically describe the underlying Code Prover analysis engine and apply to both platforms.

Configure Analysis on Desktop

Set Up Polyspace Projects on Desktop

- “Add Source Files for Analysis in Polyspace Desktop User Interface” on page 2-2
- “Contents of Polyspace Project and Results Folders” on page 2-7
- “Create Polyspace Projects from Visual Studio Build” on page 2-9
- “Create Project in Polyspace Desktop User Interface Using Configuration Template” on page 2-13
- “Update Project in Polyspace Desktop User Interface” on page 2-17
- “Modularize Large Projects in Polyspace Desktop User Interface” on page 2-20
- “Organize Layout of Polyspace Desktop User Interface” on page 2-23
- “Customize Polyspace Desktop User Interface” on page 2-25
- “Upload Results to Polyspace Access” on page 2-28

Add Source Files for Analysis in Polyspace Desktop User Interface

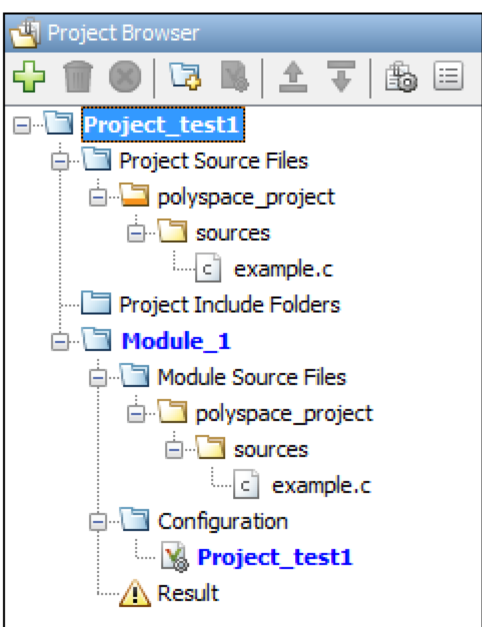
This topic shows how to create a project in the user interface of the Polyspace desktop products.

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

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

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

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



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

Polyspace Project and Source File Paths

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

For instance, suppose that a project is stored in:

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

Suppose that the project points to the source file path:

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

If you move the project to

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

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

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

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

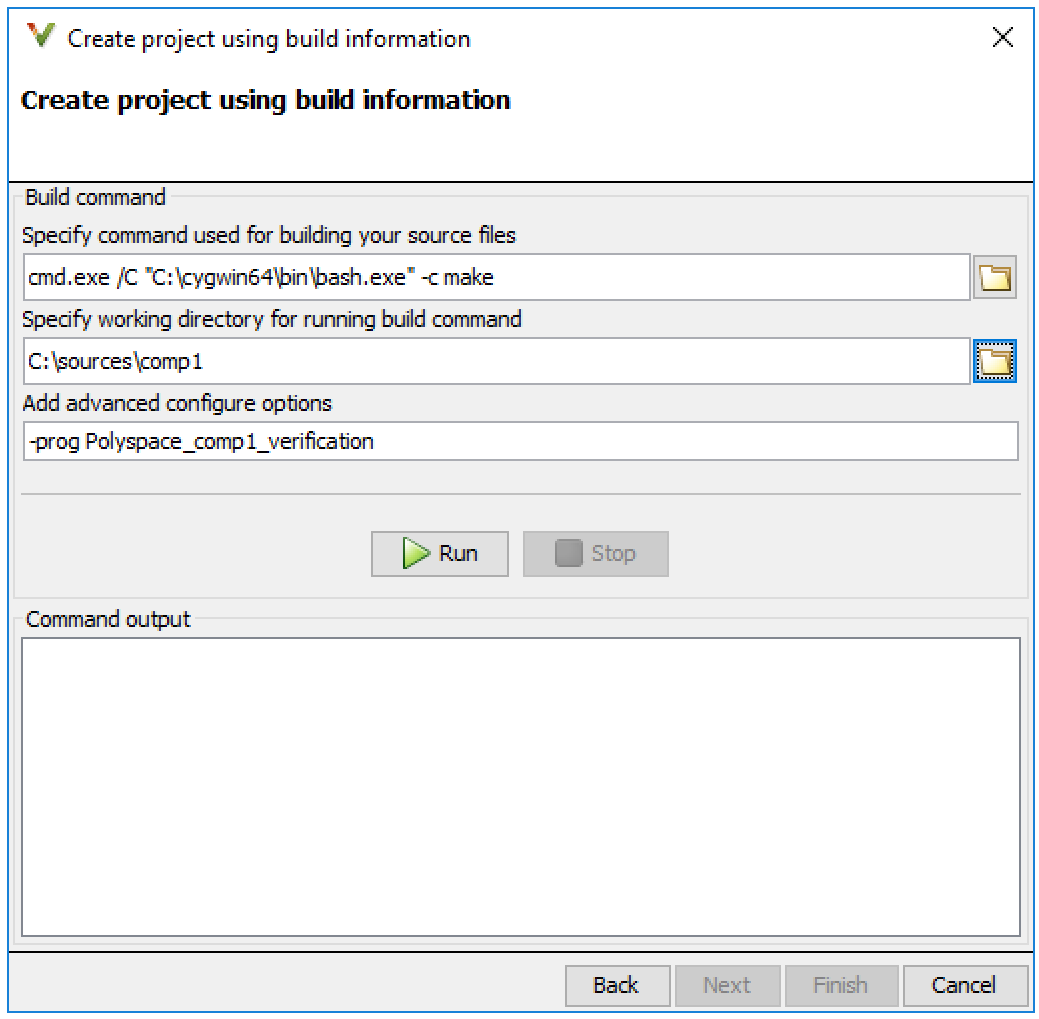
Because source file paths are recomputed relative to a project path, you can commit a Polyspace project to a version control system along with your source files. When you check out the project from your version control system and open a local copy of the project, all source file paths are recomputed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

Add Sources from Build Command

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

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

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



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

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

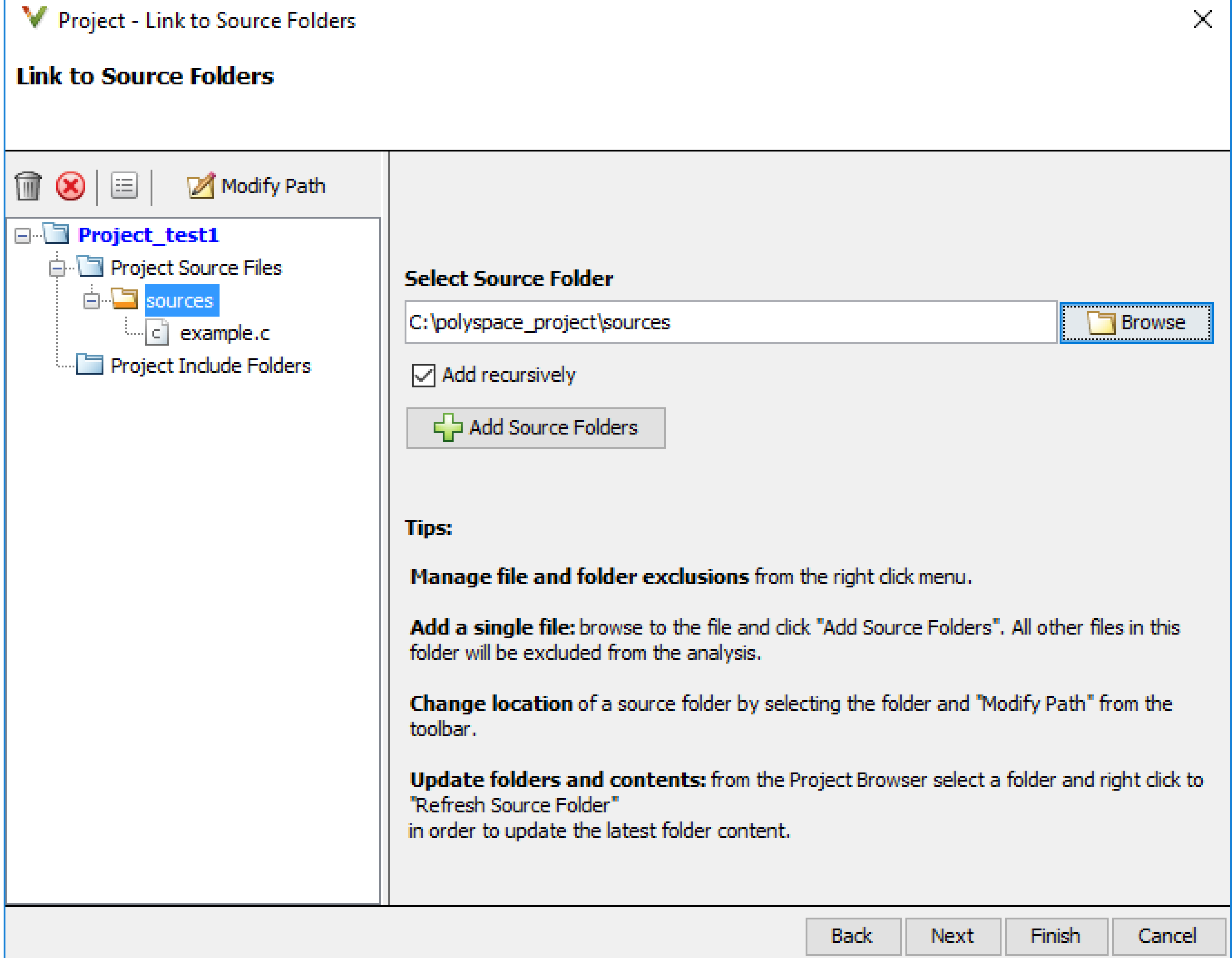
When you create a project from your build command, the **Project Browser** pane displays your source folders but not the include folders. In case you want to verify that your include folders were extracted, open the project file (with extension `.psprj`) in a text editor.

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

Add Sources Manually

Select **File > New Project**.

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



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

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

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

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

Standard Library headers, which in some special cases, might not match your compiler implementation. See also “Provide Standard Library Headers for Polyspace Analysis” on page 13-19.

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.

Add Source Files Based on AUTOSAR Design Specifications

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

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

For details, see “Run Polyspace on AUTOSAR Code” on page 8-14.

See Also

More About

- “Run Analysis in Polyspace Desktop User Interface” on page 3-2
- “Create Polyspace Projects from Visual Studio Build” on page 2-9
- “Provide Standard Library Headers for Polyspace Analysis” on page 13-19

Contents of Polyspace Project and Results Folders

This topic applies only to the Polyspace desktop products..

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

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

File Organization

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

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

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

Files in the Results Folder

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

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

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

See Also

`-results-dir`

Create Polyspace Projects from Visual Studio Build

In this section...

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

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

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

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

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

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

Create Polyspace Project from Build in Visual Studio Developer Command Prompt

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

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

   This command prompt is similar to a regular command prompt but with all Visual Studio environment variables appropriately set up.
2. Perform a full build of your Visual Studio project: at the command prompt:

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

   This step is optional. Ensuring that the build completes successfully by itself allows you to create a Polyspace project from an error-free build.
3. Run the complete build command from the previous step but prepended with the `polyspace-configure` command:

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

   For the above command to work, add the path *`polyspaceroot`*`\polyspace\bin` to the `Path` environment variable in Windows. Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\`**`R`**`2023`**`a`**.

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

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

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

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

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

  These commands generate two Polyspace projects, `AProject.psprj` and `AnotherProject.psprj`.
- Instead of creating a Polyspace project to run analysis, you can run the analysis using options files. See also “Options Files for Polyspace Analysis”. If you take the options file approach to run Polyspace, you can first run `polyspace-configure` on a Visual Studio solution to generate one options file per project in the solution.

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

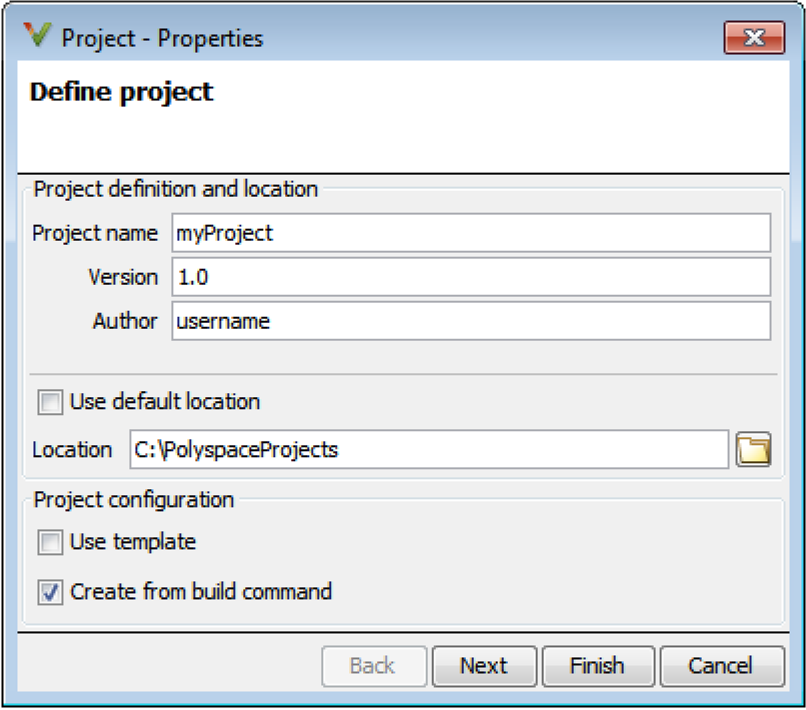
  ```
  polyspace-configure -module -output-options-path . ^
       msbuild ExampleProject.sln /t:Rebuild
  ```

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

Create Polyspace Project from Build in Visual Studio IDE

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

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



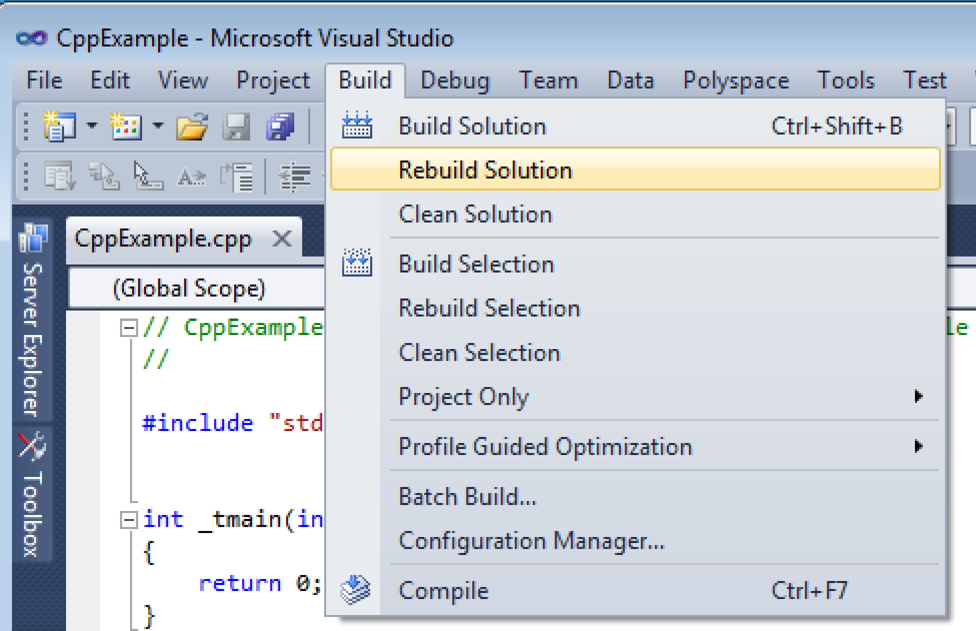
3. In the field **Specify command used for building your source files**, enter the full path to the Visual Studio executable. For instance, `"C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\devenv.exe"`.

4. In the field **Specify working directory for running build command**, enter a folder to which you have write access, for instance, `C:\temp\Polyspace`. Click Run.

   This action opens the Visual Studio environment.

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

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



6 After the project builds, close Visual Studio.

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

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

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

See Also

`polyspace-configure`

More About

- "Troubleshoot Project Creation from Visual Studio Build" on page 34-34

Create Project in Polyspace Desktop User Interface Using Configuration Template

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

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

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

Why Use Templates

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

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

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

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

Use Predefined Template

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

   If your compiler does not appear in the list of predefined templates, select **Baseline_C** or **Baseline_C++**.
4. On the next screen, add your source files and include folders.

Create Your Own Template

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

- To create a template from a project that is open on the **Project Browser** pane:

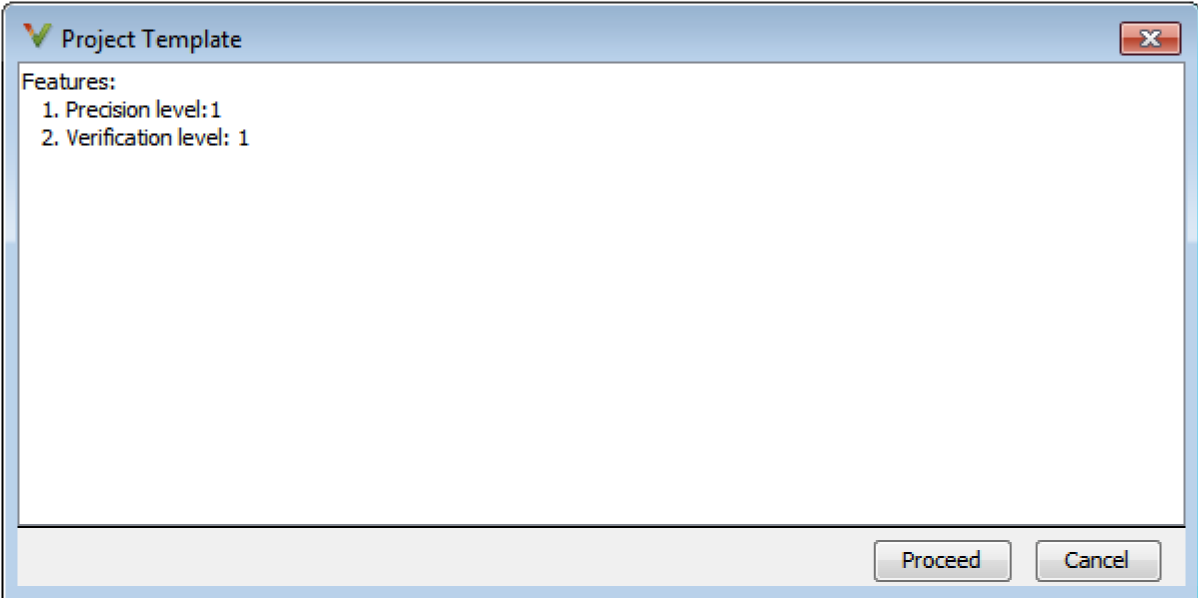
  1. Right-click the project configuration that you want to use, and then select **Save As Template**.

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

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

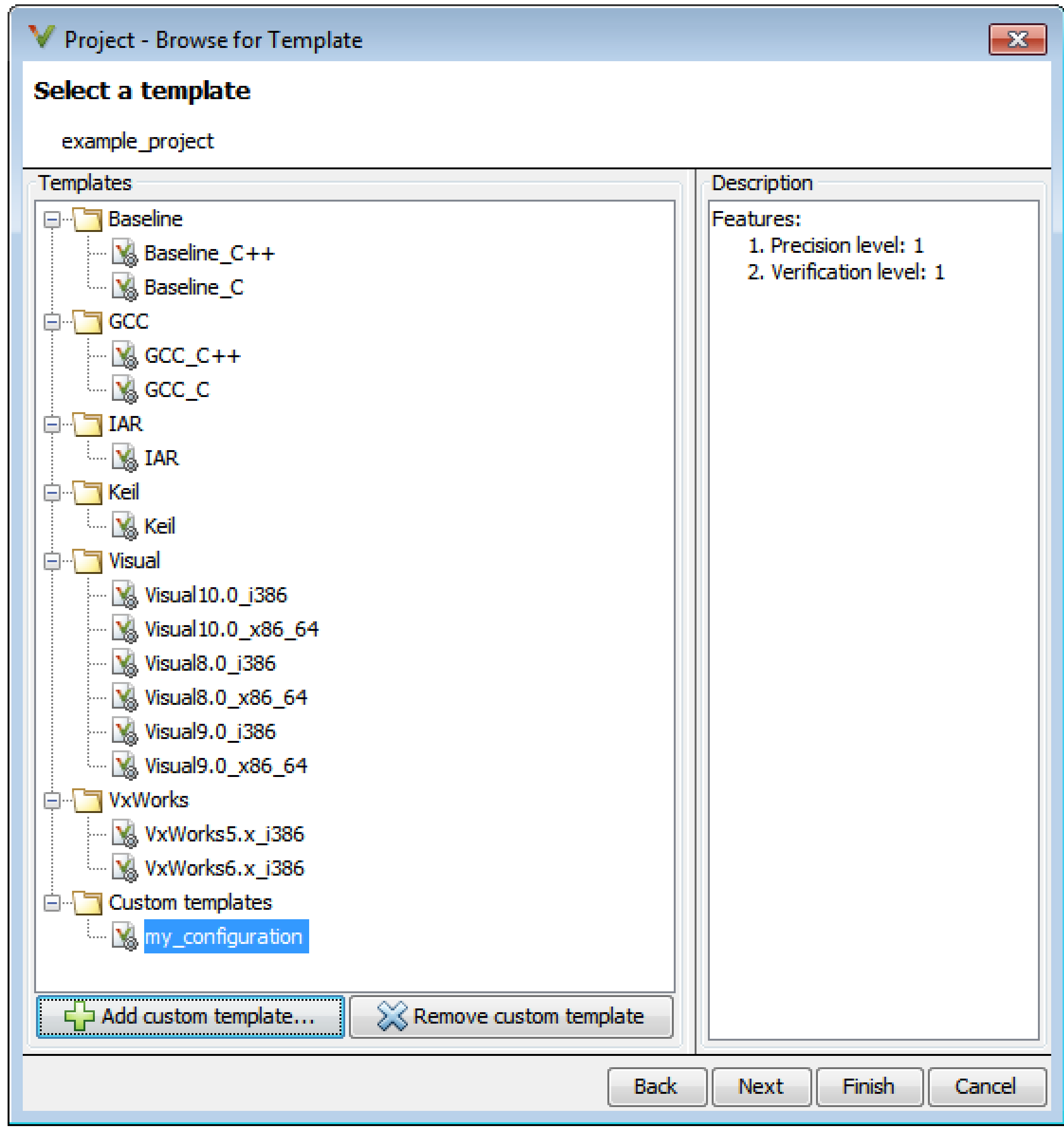
   - `Precision level (-O0 | -O1 | -O2 | -O3)`
   - `Verification level (-to)`

   You can enter this description for the template.



- When you create a new project, to use a saved template:

   1 Select Add custom template....
   2 Navigate to the template that you saved earlier, and then click **Open**. The new template appears in the **Custom templates** folder on the **Templates** browser. Select the template for use.



Sharing Project Templates

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

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

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

See Also

More About

- "Specify Polyspace Analysis Options" on page 12-2
- "Complete List of Polyspace Code Prover Analysis Options"

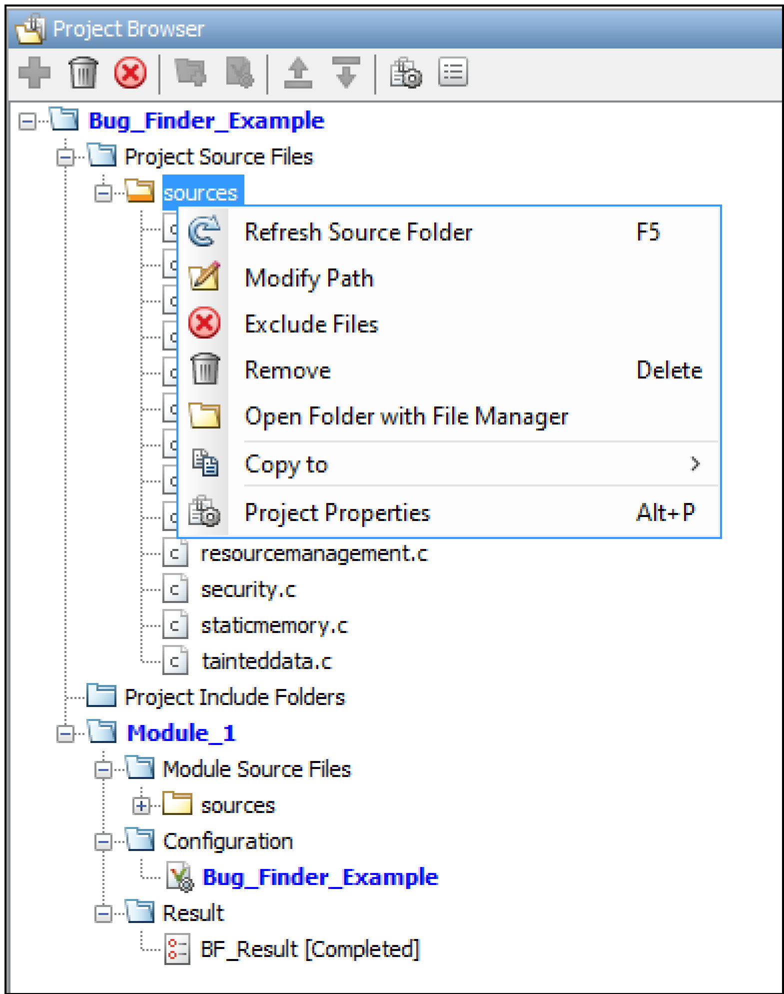
Update Project in Polyspace Desktop User Interface

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

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

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

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



Change Folder Path

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

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

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

Refresh Source List

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

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

Refresh Project Created from Build Command

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

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

Add Source and Include Folders

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

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

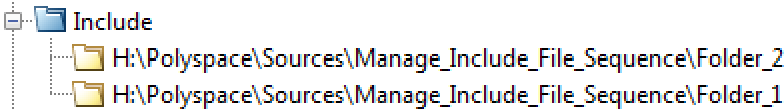
Before running an analysis, you must copy the source files to a module. Select the source files that you want to copy. To select multiple files together, press the **Ctrl** key while selecting the files. Right-click your selection. Select **Copy to** > **Module_*n***. *n* is the module number.

Manage Include File Sequence

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

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

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



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

See Also

Related Examples

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

Modularize Large Projects in Polyspace Desktop User Interface

When you add source files to a project in the Polyspace desktop user interface, by default, all files are analyzed in a single run. Analyzing the entire code base for a single application might take a long time, depending on the size of the application.

For a large application, Polyspace allows you to:

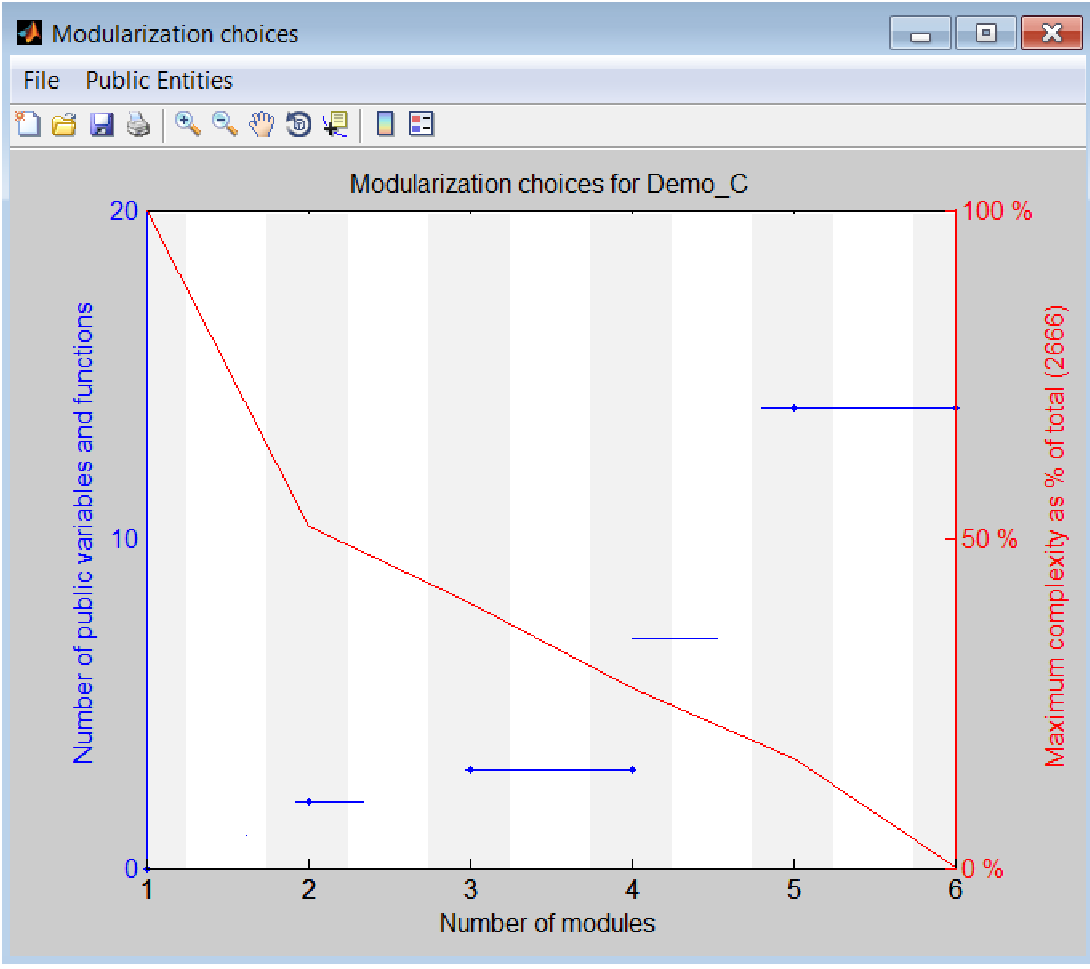
- Partition the application into modules that individually require less time to verify.
- Specify the number of modules in a trade-off between verification speed and precision.

You can carry out faster analysis with a larger number of small modules. During partitioning, the software automatically minimizes cross-module references. However, with more modules, greater cross-module referencing is required during verification, which results in a loss of precision.

To partition your application into modules:

1. Run an initial verification, which performs a limited analysis but processes all the files of your application. For example, run a verification with the following **Precision** pane settings:

   - **Precision level** — `0`
   - **Verification level** — `Software Safety Analysis level 0`
2. In the **Project Browser** view, select the results folder.
3. Select **Tools** > **Run Modularize**. The software analyzes your application code and displays two plots in a new Modularization choices window.



The plots show the following information:

- Red — Maximum complexity of a module versus number of modules, which is expressed as a percentage of the total complexity of the application.
- Blue — Number of public variables and functions when modules are limited by a given complexity.

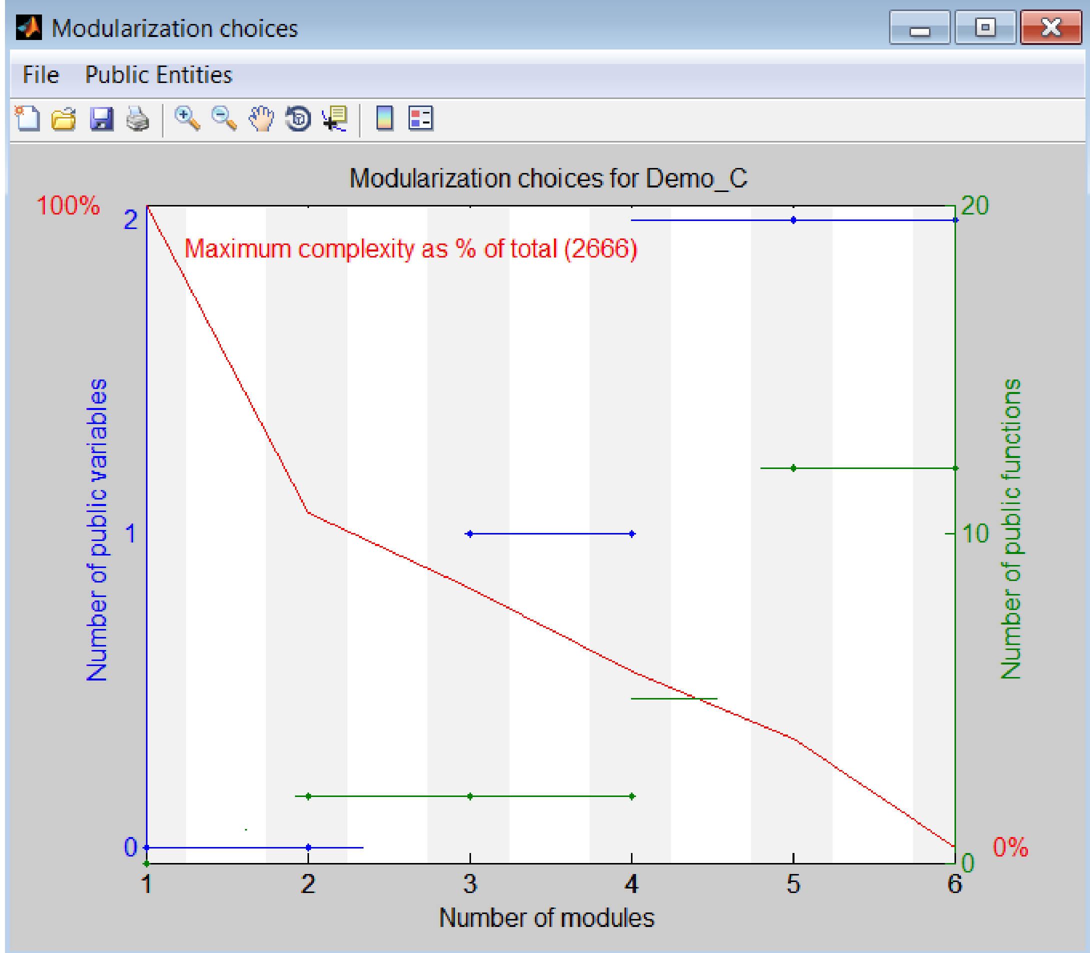
4 From the plots, identify the number of modules into which your application must be partitioned. In this example, a suitable number is 2 or 4.

The number of partitioned modules that you choose involves a trade-off between the following:

- Time — The smaller the maximum complexity, the shorter the time required for verification. This time saving is even greater if the different modules are verified in parallel.
- Precision — The smaller the number of public variables and functions, the greater the precision of the verification.

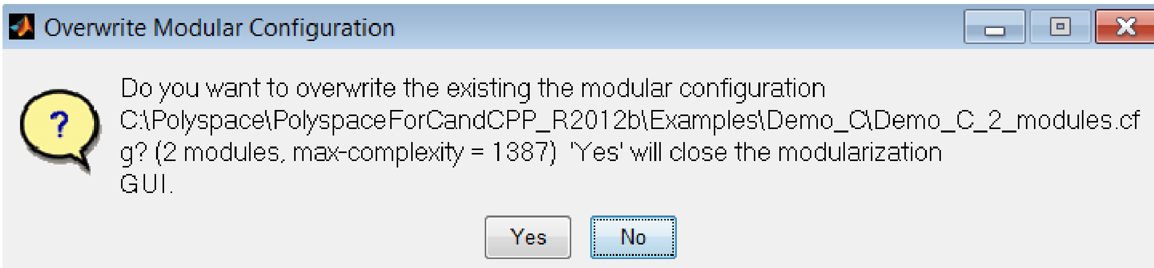
Select a number just after a big drop in maximum complexity and before a big increase in the number of public functions and variables. The precision of a modular verification can be very sensitive to the number of public variables. If the series of horizontal blue lines ascends so gradually that there is no clear number choice, then:

a On the toolbar, select **Public Entities** > **Separate functions and variables**. The software displays the number of public variables and functions separately.



b Select a point just before a big jump in the number of public variables. In this example, you must click the gray region associated with 2.

5 Click the vertical gray region associated with the number of modules that you choose, for example, 2. A dialog box opens.



6 Click **Yes**. The software generates a new project with two modules containing the partitioned code.

Organize Layout of Polyspace Desktop User Interface

The Polyspace user interface has two default layouts of panes.

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

<table>
<tr><td rowspan="2">Project Browser</td><td>Configuration</td></tr>
<tr><td>Output Summary</td></tr>
</table>

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

<table>
<tr><td rowspan="2">Results List</td><td>Result Details</td></tr>
<tr><td>Dashboard</td></tr>
</table>

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

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

Create Your Own Layout

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

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

To move a pane to another location:

1. Float the pane in one of three ways:

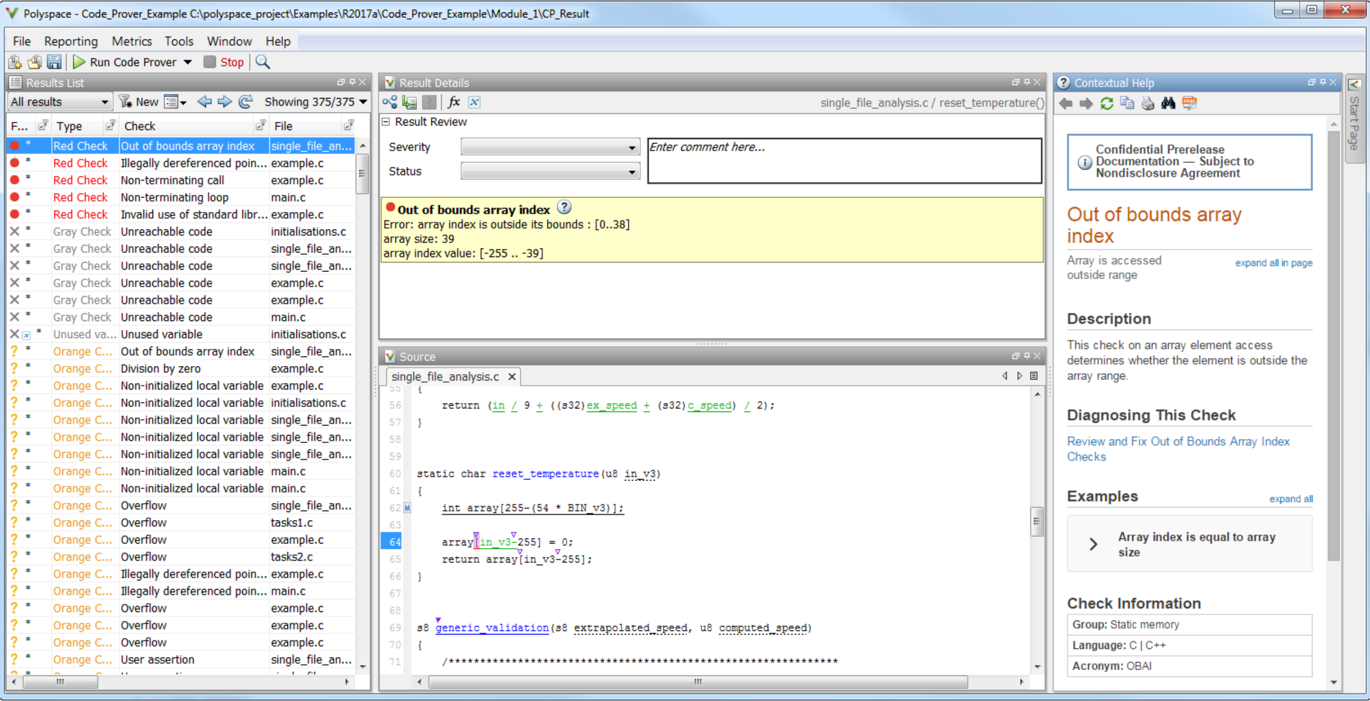
   - Click and drag the blue bar on the top of the pane to float all tabs in that pane.

     For instance, if **Project Browser** and **Results List** are tabbed on the same pane, this action floats the pane together with its tabs.
   - Click and drag the tab at the bottom of the pane to float only that tab.

     For instance, if **Project Browser** and **Results List** are tabbed on the same pane, dragging out **Project Browser** creates a pane with only **Project Browser** on it and floats this new pane.
   - Click on the top right of the pane to float all tabs in that pane.
2. Drag the pane to another location until it snaps into a new position.

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

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



Save and Reset Layout

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

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

See Also

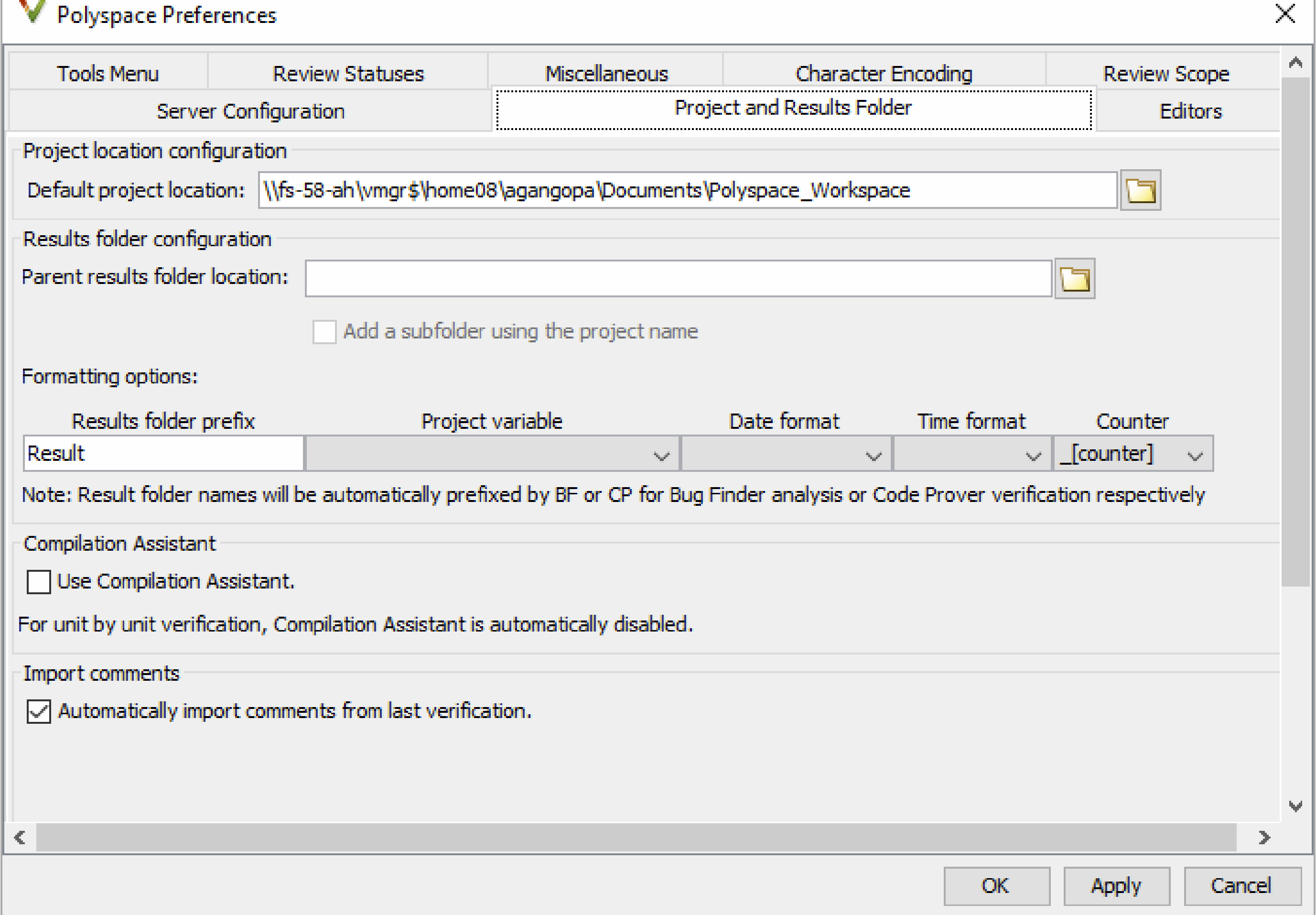
More About

- "Customize Polyspace Desktop User Interface" on page 2-25

Customize Polyspace Desktop User Interface

In this section...
"Possible Customizations" on page 2-25
"Storage of Polyspace User Interface Customizations" on page 2-27

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



Possible Customizations

Change Default Font Size

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

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

  For example, to increase the default size by 1 point, select +1.
- To increase the font size of the code on the **Source** pane and the **Code Editor** pane, select a value for **Source code font size**.

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

Specify External Text Editor

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

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

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

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

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

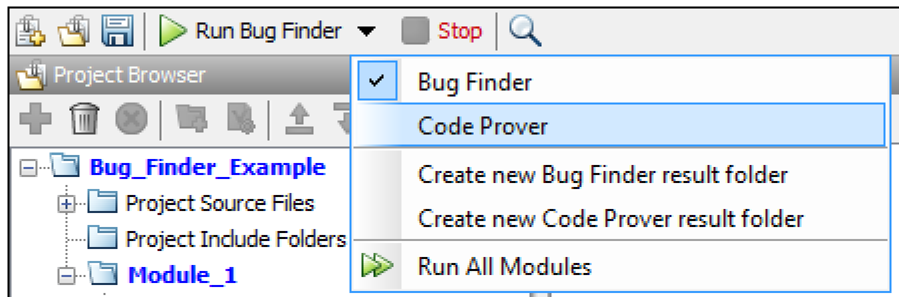
For console-based text editors, you must create a terminal. For example, to specify `vi`:

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

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

Create Naming Convention for Results Folder

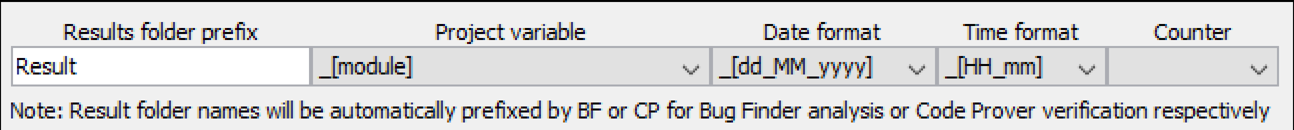
By default, results are stored in a subfolder of the project folder. When you run an analysis, you can overwrite the results of the previous run or create a new results folder.



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

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

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



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

Create Custom Review Status

When reviewing Polyspace results, you can assign a status such as `To fix` or `Justified`. See "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" on page 23-2.

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

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

   Optionally, to specify that Polyspace should consider results with this review status justified, select the checkbox next to the **Add** button. See also "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" on page 23-2.

Storage of Polyspace User Interface Customizations

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

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

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

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

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

Upload Results to Polyspace Access

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

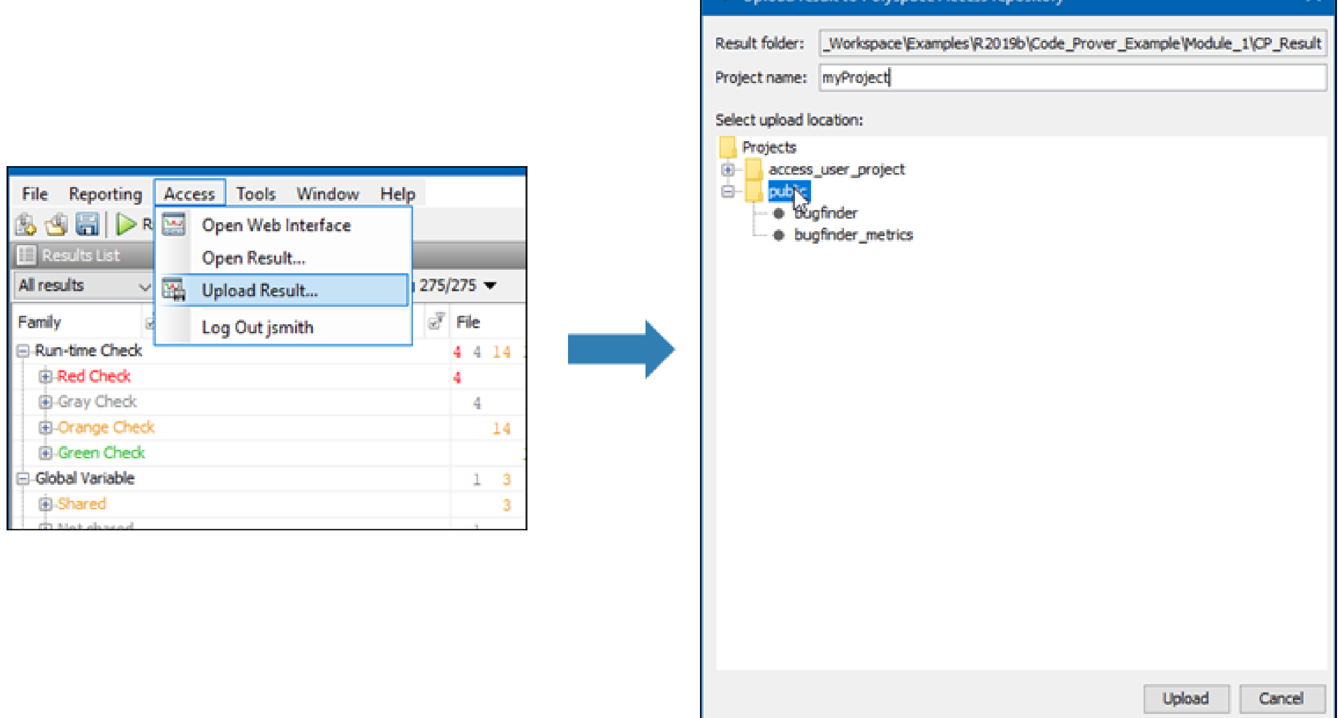
Polyspace assigns a unique run ID to each analysis run that you upload and increments the run ID with each upload to any project. If you use an automation tool such as Jenkins to upload results, the Polyspace Access run ID is not related to the tool job ID.

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

Upload Results from Polyspace Desktop Client

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

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



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

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

Once you save the changes you make to these fields in the desktop interface, the changes are reflected in the Polyspace Access web interface. To create custom statuses, see "Add Custom Status in Polyspace Access Project" on page 27-3.

Upload Results at Command Line

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

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

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

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

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

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

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

Results Upload Compatibility and Permissions

Results Compatibility

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

If you upload results generated with a Polyspace product version R2018b or earlier, you cannot view these results in the Polyspace Access **REVIEW** perspective. To review R2018b or earlier results that you uploaded to Polyspace Access, see "Open Polyspace Access Results in a Desktop Interface" on page 29-2.

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

User Permissions for Uploaded Results

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

Results that you upload to the **public** folder are visible to all Polyspace Access users. For more information, see “Manage Project Permissions” on page 28-3.

See Also

`polyspace-access`

More About

- “Register Polyspace Desktop User Interface”

Run Polyspace Analysis on Desktop

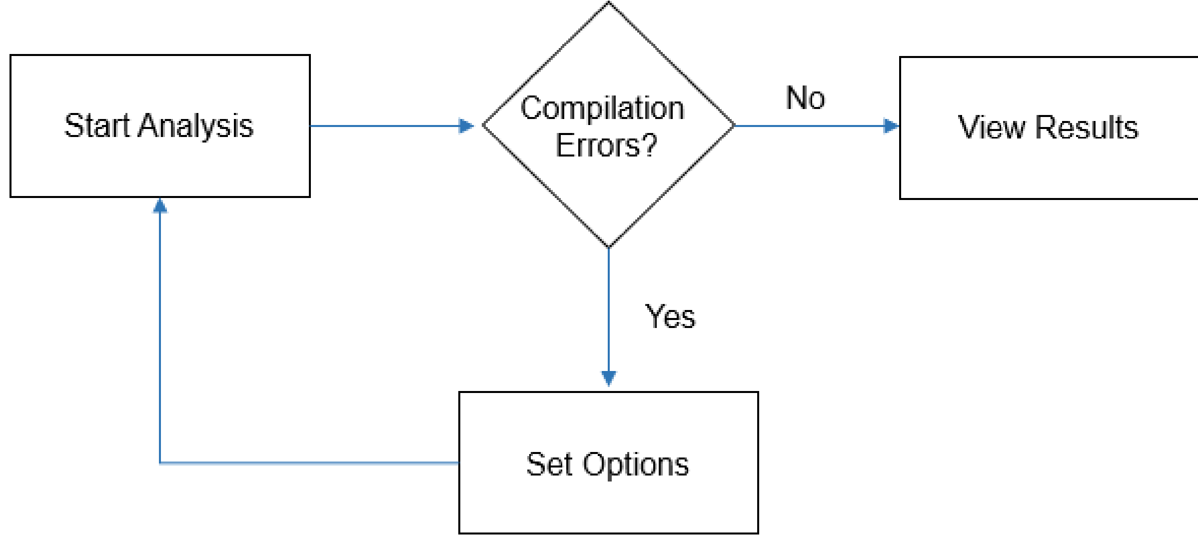
Run Analysis in Polyspace Desktop User Interface

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

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

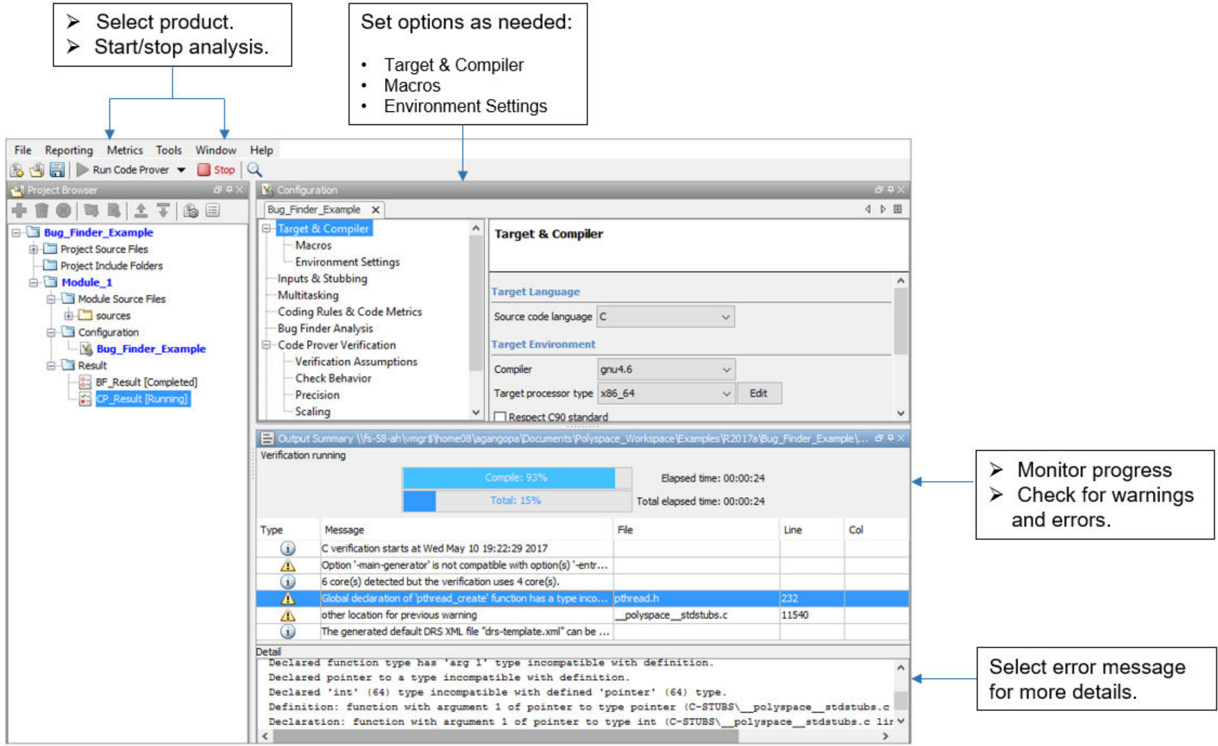
This topic describes how to run an analysis in the Polyspace user interface, monitor progress, fix compilation issues, and open analysis results as available.

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



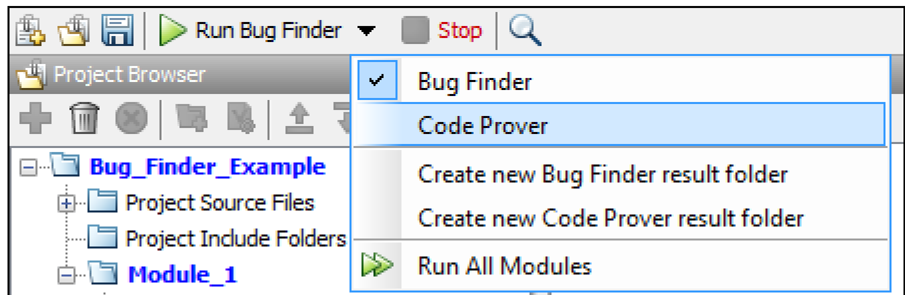
Arrange Layout of Windows for Project Setup

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



Set Product and Result Location

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



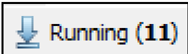
The results are stored in subfolders `Module_1`, `Module_2`, and so on in the project folder. To find the physical location of the project folder, right-click a project on the **Project Browser** pane and select **Open Folder with File Manager**.

To use a different folder naming convention or a different storage location for results, select **Tools > Preferences** and use the options on the **Project and Results Folder** tab. See also “Create Naming Convention for Results Folder” on page 2-26.

Start and Monitor Analysis

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

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

  The icon indicates the number of results available. To open the results, click the icon. Once the analysis is over, the **Running** label in the icon changes to **Completed**. To reload the full set of results, click the icon again.
- Code Prover: You can see results only after the analysis is complete. Code Prover is more likely to report compilation errors because it does a more rigorous analysis and must follow stricter rules for compilation. The progress bar distinguishes between the various phases of analysis starting from compilation.

Fix Compilation Errors

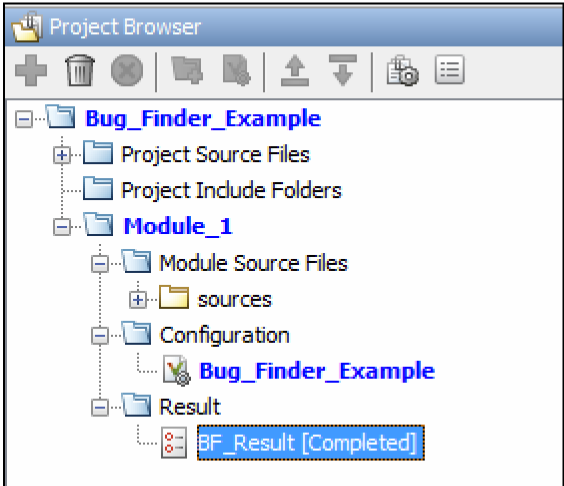
If compilation errors occur, the analysis continues on the remaining files that do compile. The **Dashboard** pane shows that some files did not compile and links to the **Output Summary** pane for details. The **Output Summary** pane shows compilation errors with a icon.

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

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

Open Results

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



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

See Also

More About

- “Run Polyspace Analysis from Command Line” on page 4-2
- “Run Polyspace Analysis by Using MATLAB Scripts” on page 5-9
- “Review Polyspace Code Prover Results in Polyspace User Interface”
- “Upload Results to Polyspace Access” on page 2-28

Storage of Temporary Files During Polyspace Analysis

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

- "Fix Polyspace Errors Related to Temporary Files" on page 34-91
- "Reduce Memory Usage and Time Taken by Polyspace Analysis" on page 34-10

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

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

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

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

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

Run Polyspace Analysis with Windows or Linux Scripts

- “Run Polyspace Analysis from Command Line” on page 4-2
- “Modularize Polyspace Analysis by Using Build Command” on page 4-5
- “Select Files for Polyspace Analysis Using Pattern Matching” on page 4-11
- “Modularize Polyspace Analysis at Command Line Based on an Initial Interdependency Analysis” on page 4-15
- “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 4-17

Run Polyspace Analysis from Command Line

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

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

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

Specify Sources and Analysis Options Directly

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

For instance:

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

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

  If your current folder contains a `sources` subfolder with the source files, you can omit the `-sources` flag. The analysis considers files in `sources` and all subfolders under `sources`.
- To specify the target processor, use the `-target` option. For instance, to specify the `m68k` processor for your source file `file.c`, use the command:

  ```
  polyspace-bug-finder -sources "file.c" -lang c -target m68k
  ```
- To check for violation of MISRA C™ rules, use the `-misra2` option. For instance, to check for only the required MISRA C rules on your source file `file.c`, use the command:

  ```
  polyspace-bug-finder -sources "file.c" -misra2 required-rules
  ```
- To specify a results folder, use the option `-results-dir`.

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

For the full list of analysis options, see:

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

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

```
polyspace-code-prover -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 `listofoptions.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 `listofoptions.txt`.

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

See also `-options-file`.

Create Options File from Build System

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

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

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

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

   See also `polyspace-configure`.

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

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

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

3. Open the results in the Polyspace user interface.

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

See Also

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

More About

- “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 4-17

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

External Websites

- Set up Continuous Code Verification with Jenkins

Modularize Polyspace Analysis by Using Build Command

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

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

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

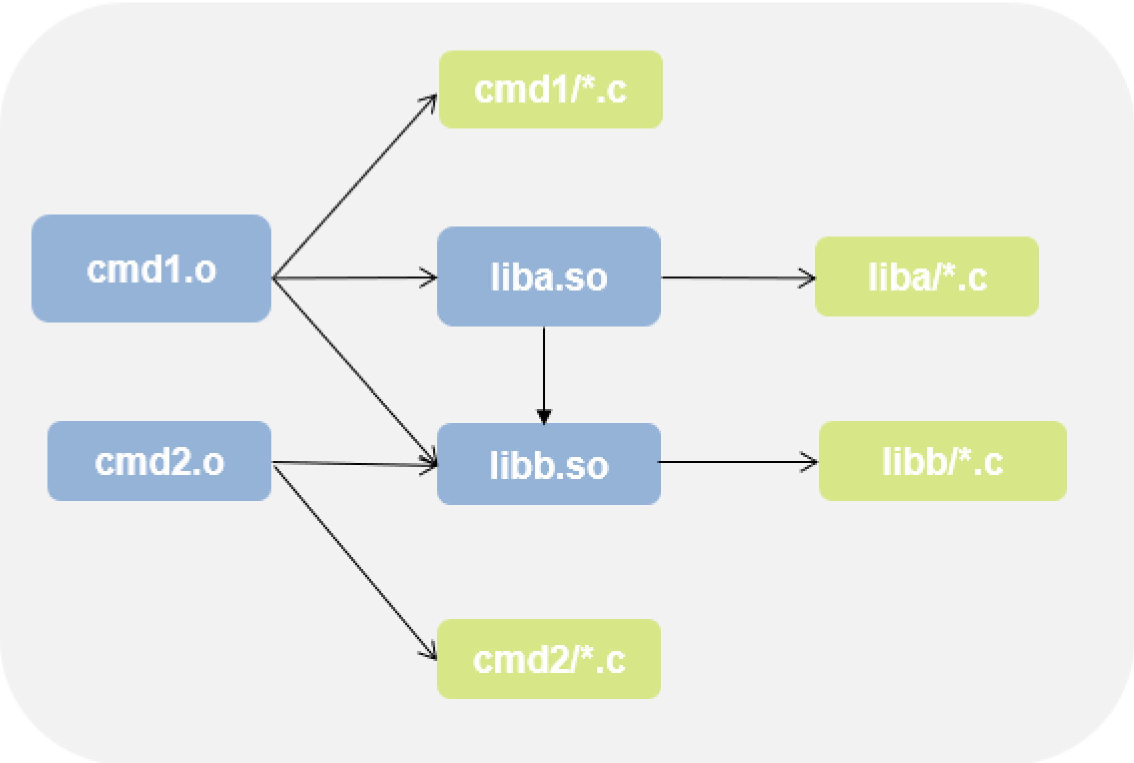
> To try this example, use the files in *`polyspaceroot`*`\help\toolbox\codeprover\examples\multiple_modules`. Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2023a` or `C:\Program Files\Polyspace Server\R2023a`.

Build Source Code

Inspect the makefile. The makefile creates four binaries:

```
CC := gcc

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

all: cmd1 cmd2

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

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

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

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

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

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



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

```
make -B
```

Make sure that the build runs to completion.

Create One Polyspace Options File for Full Build

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

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

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

```
polyspace-code-prover -options-file psoptions -results-dir results
```

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

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

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

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

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

Create Options File for Specific Binary in Build Command

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

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

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

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

```
polyspace-code-prover -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" on page 18-6.

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-code-prover -options-file optionsFilesFolder\cmd1.psopts -results-dir results
```

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

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” on page 18-6.

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

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

See Also

`polyspace-configure` | `polyspace-code-prover` | `polyspace-code-prover-server`

More About

- "Run Polyspace Analysis from Command Line" on page 4-2
- "Create Polyspace Analysis Configuration from Build Command (Makefile)" on page 13-21

Select Files for Polyspace Analysis Using Pattern Matching

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

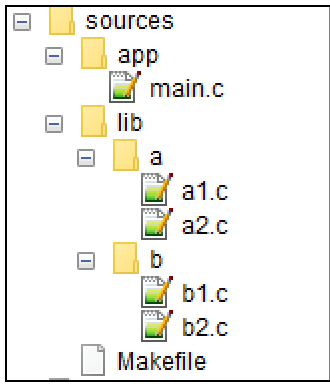
When to Specify File Selection Patterns

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

Select Files When Setting Up Polyspace Analysis from Build Command

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

This folder structure applies to these examples.



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

Run this command:

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

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

Select Files When Running Bug Finder Analysis

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

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

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

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

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

For more information, see:

- `-classification`
- "Classify Project Files Into File Sets for Precise Control of Bug Finder Analysis"

Supported Patterns for File Selection

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

Glob Pattern Syntax	Example
No special characters, slashes ('/'), or backslashes ('\'). Pattern matches corresponding files, but not folders.	`-include-sources "main.c"` matches: `/sources/app/main.c`

Pattern contains `'*'` or `'?'` special characters.
 	
`'*'` matches zero or more characters in file or folder name.
 	
`'?'` matches one character in file or folder name.
 	
The matches do not include path separators.
	`-include-sources "b?.c"` matches:
	
	`/sources/lib/b/b1.c`
	
	`/sources/lib/b/b2.c`
	
	`-include-sources "app/*.c"` matches:
	
	`/sources/app/main.c`

Pattern starts with: • A slash `'/'` (UNIX). • Drive letter, for example `C:\` (Windows). Pattern matches absolute path only.	`-include-sources "/a"` does not match anything. `-include-sources "/sources/app"` matches: `/sources/app/main.c`

Glob Pattern Syntax	Example

Pattern ends with:
• A slash (UNIX).
• A backslash (Windows).
• A double asterisk (`'**'`)
Pattern matches all files under specified folder.
`'**'` is ignored if it is at the start of the pattern.
	`-include-sources "a/"` matches
	`/sources/lib/a/a1.c`
	`/sources/lib/a/a2.c`

Pattern contains:
• `'/**/'` (UNIX).
• `'**\'` (Windows).
Pattern matches zero or more folders in the specified path.
	`-include-sources "lib/**/?1.c"` matches:
	`/sources/lib/a/a1.c`
	`/sources/lib/b/b1.c`

Pattern starts with `'.'` or `'..'`.
Pattern matches paths relative to the path where you run the command.
	If you start `polyspace-configure` from `/sources/lib/a`,
	`-include-sources "../lib/**/b?.c"` matches:
	`/sources/lib/b/b1.c`
	`/sources/lib/b/b2.c`

Pattern is a UNC path on Windows .	If your files are on server `myServer`:
	`\\myServer\sources\lib\b**` matches:
	`\\myServer\sources\lib\b\b1.c`
	`\\myServer\sources\lib\b\b2.c`

`polyspace-configure` does not support these glob patterns:

- Absolute paths relative to the current drive on Windows.

  For instance, `\foo\bar`.
- Relative paths to the current folder.

  For instance, `C:foo\bar`.
- Extended length paths in Windows.

  For instance, `\\?\foo`.
- Paths that contain '.' or '..' except at the start of the pattern.

  For instance, `/foo/bar/../a?.c`.
- The `'*'` character by itself.

See Also
`-classification` | `polyspace-configure`

More About
- "Classify Project Files Into File Sets for Precise Control of Bug Finder Analysis"

Modularize Polyspace Analysis at Command Line Based on an Initial Interdependency Analysis

Analyzing the entire code base for a single application might take a long time, depending on the size of the application.

For a large application, Polyspace allows you to:

- Partition the application into modules that individually require less time to verify.
- Specify the number of modules in a trade-off between verification speed and precision.

You can carry out faster analysis with a larger number of small modules. During partitioning, the software automatically minimizes cross-module references. However, with more modules, greater cross-module referencing is required during verification, which results in a loss of precision.

Basic Options

You can partition an application into modules using the following batch command:

```
polyspace-modularize [target_folder] {option1,option2,...}
```

This table describes the basic options that you can use.

Option	Description
target_folder	Folder that contains the results of the initial run that processes source files. Default is the folder from which you run `polyspace-modularize`.
`-o=`*output_folder*	Output folder for partitioned application. Default is the folder from which you run `polyspace-modularize`.
`-gui=`*max_n*	The Polyspace verification environment displays the Modularizing choices window with a predefined limit for the maximum number of modules that you can select. Use this option to specify a new limit *max_n*.
`-matlab=`*max_n*	If data cache for Modularizing choices window does not exist, create cache *project_name_max_n*`.m`. Cache enables faster display of Modularizing choices window. *project_name* is the value used by `-prog` option. *max_n* is the limit for the maximum number of modules that you can select. No action if cache already exists.
`-modules=`*n*	Partition application into *n* modules. Identical to clicking the gray region associated with *n* in the Modularizing choices window.

Option	Description
-max-complexity=*max_c*	Partitions application into modules with reference to specified maximum complexity *max_c*. The complexity of a function is a number that is related to the size of the function. The complexity of a module is the sum of the complexities of the functions in the module. When partitioning your application, the software minimizes the use of cross-module references to functions and variables, subject to the constraint that the complexity of a module does not exceed *max_c*. If you make *max_c* sufficiently large, the software generates only one module, which is identical to the original, unpartitioned application.

Constrain Module Complexity During Partitioning

To force all functions to have a complexity of 1, run the following command:

```
polyspace-modularize -uniform-complexities
```

Result Folder Names

By default, modularization results folders are named *projectName_kk_module*:

- *kk* is either the max complexity argument you give to `-max-complexity`, or the number of modules.

You can control the naming of result folders in the ith module using the `-stem` option:

```
polyspace-modularize -stem=stem_format
```

stem_format is a string. The `#` and `@` characters in the string are processed as follows:

- `#` — Replaced by the number of modules in the partitioning.
- `@` — Replaced by the argument of `-max-complexity`.

For example, if you want a specific name, `MyName`, which overrides the project name and does not incorporate the module number, then run:

```
polyspace-modularize -stem=MyName
```

Forbid Cycles in Module Dependence Graph

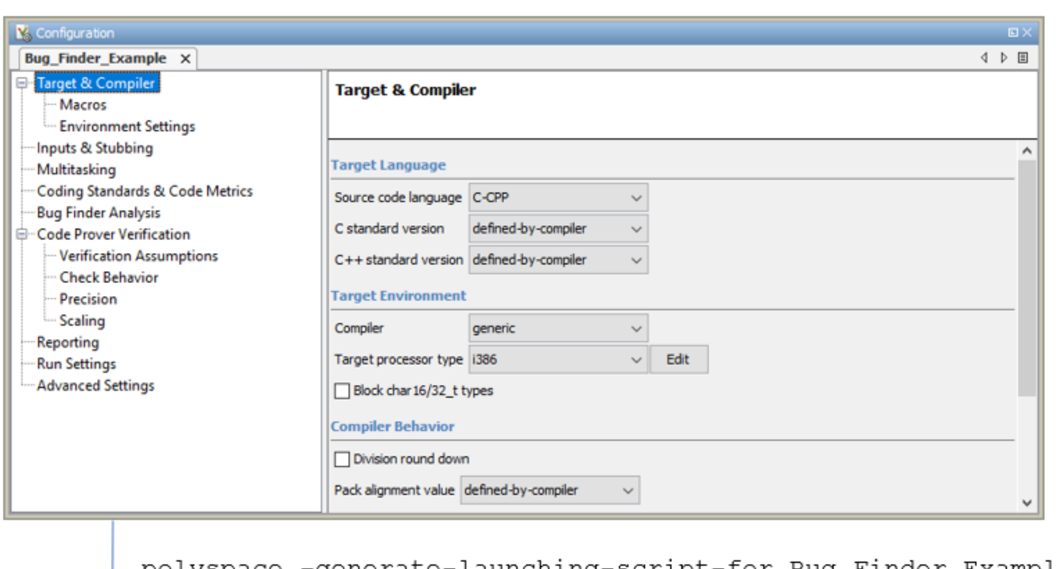
By default, the software allows the module dependence graph to have cycles. However, in some cases, you might get better results with acyclic graphs. Use the following command:

```
polyspace-modularize -forbid-cycles
```

Configure Polyspace Analysis Options in User Interface and Generate Scripts

In this section...
“Prerequisites” on page 4-18
“Generate Scripts from Configuration” on page 4-18
“Run Analysis with Generated Scripts” on page 4-19

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

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

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

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

Prerequisites

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

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

Generate Scripts from Configuration

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

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

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

   See "Add Source Files for Analysis in Polyspace Desktop User Interface" on page 2-2.
2. Specify the analysis options on the **Configuration** pane in the Polyspace project. To open this pane, in the project browser, click the configuration node in your Polyspace project.

   See "Specify Polyspace Analysis Options" on page 12-2.
3. Run the analysis. Based on compilation errors and analysis results, modify options as needed.

   See "Run Analysis in Polyspace Desktop User Interface" on page 3-2.
4. Once your analysis options are set, generate a script from the project (`.psprj` file).

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

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

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

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

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

These files are generated for scripting the analysis:

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

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

Run Analysis with Generated Scripts

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

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

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

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

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

   The scripts refer to the server product executable instead of the desktop products.
3. Execute the script `launchingCommand.bat` or `launchingCommand.sh` at periodic intervals or based on predefined triggers.

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

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

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

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

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

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

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

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

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

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

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

See Also

`-generate-launching-script-for`

Related Examples

- "Run Polyspace Analysis from Command Line" on page 4-2
- "Run Polyspace Code Prover on Server and Upload Results to Web Interface"

Run Polyspace Analysis with MATLAB Scripts

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

Integrate Polyspace with MATLAB and Simulink

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

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

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

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

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

Same Release of Polyspace and MATLAB

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

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

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

Prerequisite

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

Integrate Polyspace with MATLAB or Simulink

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

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

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

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

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

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

3. Restart MATLAB.

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

Unlink and Relink MATLAB and Polyspace

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

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

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

MATLAB Release Earlier Than Polyspace

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

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

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

Prerequisite

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

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

Integrate Polyspace with Cross-Release MATLAB or Simulink

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

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

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

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

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

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

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

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

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

Check Integration Between MATLAB and Polyspace

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

```
ver
```

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

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

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

See Also
`polyspacesetup`

More About
- “Polyspace Support of MATLAB and Simulink from Different Releases” on page 6-68
- “Code Prover Analysis with MATLAB Scripts”
- “Code Prover Analysis in Simulink”
- “Code Prover Analysis in MATLAB Coder”
- “Fix Issues When when Integrating Polyspace with MATLAB and Simulink” on page 34-73

Get Started with Polyspace Analysis by Using MATLAB

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

Prerequisites

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

Run Polyspace Analysis by Using MATLAB

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
9×4 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
`fopen`	Opens a file for binary read access. For instance, use this function to read an error log file.
`fclose`	Closes a file that was opened by using `fopen`. For instance, use this function to close an error log file after reading it.
`open`	Opens a file outside MATLAB in an appropriate application. For instance, use this function to open `psprj` files in the Polyspace UI.

Function	Application
`exist`	Checks for the existence of an entity. For instance, use this function to check if a particular folder or file already exists.
`delete`	Deletes a file or an object. For instance, use this function to delete older results or unnecessary options objects.
`questdlg`	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.
`clear`	Clears the workspace by deleting all objects. You can this function at the beginning of the Polyspace analysis.
`clc`	Clears all text from the MATLAB Command Window.
`fullfile`	Builds full file names from its parts. For instance, use this function to construct the full paths to source files.
`char`	Converts an array to a character array. For instance, use this function to construct the input arguments to functions that take character arrays.
`string`	Converts a variable into string arrays. For instance, use this function to construct input arguments for functions that take strings.
`dir`	Lists the content of the current working folder. For instance, use this function to find specific files or folders in the current folder.
`system`	Executes operating system commands and returns their outputs. For instance, use this function to execute a command-line script without exiting MATLAB.
`disp`	Displays the value of the input variable. For instance, use this function for debugging code, similar to how `printf()` is used in C code.
`visdiff`	Compares two files or folder. For instance, use this function to compare results from different Polyspace analysis to see the difference.
`ismember`	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.
`any`	Determines if any array elements are nonzero. For instance, use this function to check for new results.
`nnz`	Returns the number of nonzero matrix elements. For instance, use this function to check for new results.
`fieldnames`	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` | `polyspaceCodeProver` | `run` | `run`

Related Examples

- "Run Polyspace Analysis by Using MATLAB Scripts" on page 5-9
- "Visualize Code Prover Analysis Results in MATLAB" on page 25-14
- "Troubleshoot Polyspace Analysis from MATLAB" on page 5-18
- "Generate MATLAB Scripts from Polyspace User Interface" on page 5-16

Run Polyspace Analysis by Using MATLAB Scripts

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

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

```
proj = polyspace.Project

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

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

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

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

See also `polyspace.Project`.

Prerequisites

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

Specify Multiple Source Files

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

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

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

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

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

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

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

You can explicitly exclude files from analysis. For instance:

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

% Specify files to exclude.
file1 = fullfile(sourceFolder,'security.c');
file2 = fullfile(sourceFolder,'tainteddata.c');
proj.Configuration.InputsStubbing.DoNotGenerateResultsFor = ['custom=' file1 ...
     ',' file2];
```

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

Check for MISRA C:2012 Violations

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

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

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

```
% Enable MISRA C checking
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = 'mandatory';

% Disable defect checking
proj.Configuration.BugFinderAnalysis.EnableCheckers = false;

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

% Read summary of results
resObj = proj.Results;
misraSummary = getSummary(resObj, 'misraC2012');
```

Check for Specific Defects or Coding Rule Violations

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

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

```
% Disable rules
misraRules = polyspace.CodingRulesOptions('misraC2012');

misraRules.Section_8_Declarations_and_definitions.rule_8_1 = false;
misraRules.Section_8_Declarations_and_definitions.rule_8_2 = false;
```

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

% Configure analysis
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = misraRules;
```

See also `polyspace.CodingRulesOptions`.

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

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

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

Find Files That Do Not Compile

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

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

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

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

Run Analysis on Server

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

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

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

Specify the head node that manages the Polyspace jobs:

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

Run analysis as usual.

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

Open the results from the results folder location.

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

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

See Also

polyspace.Project | polyspaceCodeProver | -scheduler

Related Examples

- “Generate MATLAB Scripts from Polyspace User Interface” on page 5-16
- “Visualize Code Prover Analysis Results in MATLAB” on page 25-14
- “Troubleshoot Polyspace Analysis from MATLAB” on page 5-18

Compare Results from Different Polyspace Runs by Using MATLAB Scripts

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

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

Review Only New Results Compared to Last Run

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

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

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

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

```
proj = polyspace.Project;

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

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

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

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

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

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

The key functions used in this example are:

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

Review New Results and Unreviewed Results from Last Run

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

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

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

with this section:

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

See Also

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

More About

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

Generate MATLAB Scripts from Polyspace User Interface

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

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

Prerequisites

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

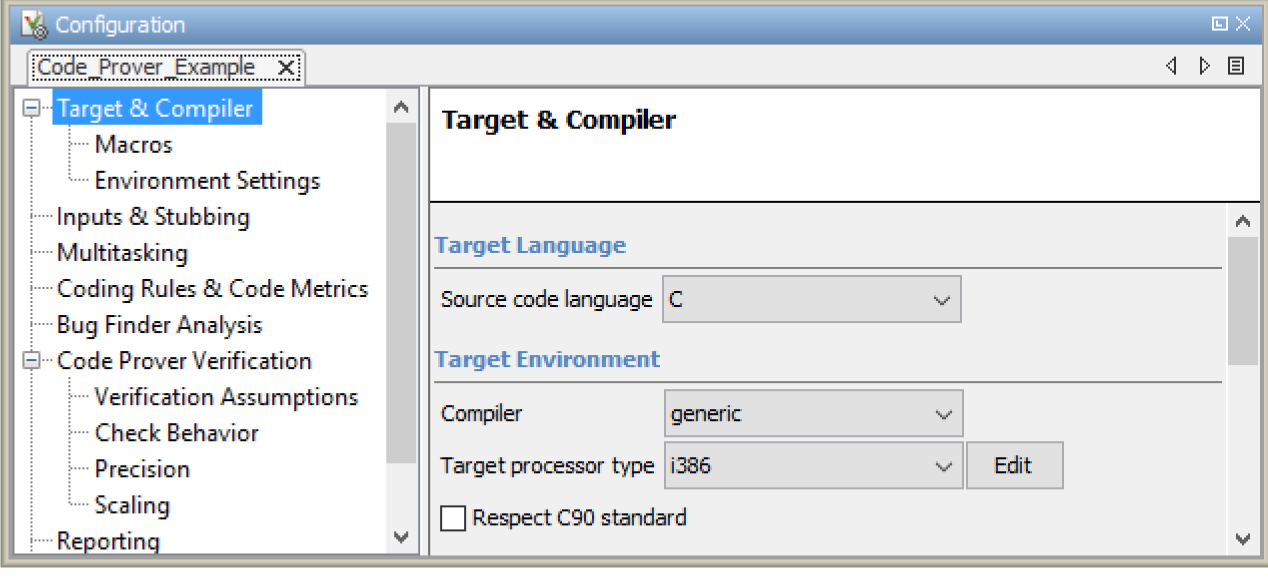
Create Scripts from Polyspace Projects

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

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

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

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



1 Open MATLAB.

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

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

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

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

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

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

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

   ```
   run(filePath);
   ```

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

See Also

Related Examples

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

Troubleshoot Polyspace Analysis from MATLAB

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

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

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

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

Prerequisites

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

Capture Polyspace Analysis Errors in Error Log

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

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

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

The function is defined as follows.

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

    proj = polyspace.Project;

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

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


    % Run analysis
```

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

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

    % Check log file for compilation errors
    numErrors = 0;

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

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

    fclose(errorFile);

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

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

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

See Also

`polyspace.Project`

Related Examples

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

Run Polyspace Analysis in Simulink

Run Polyspace Analysis on Code Generated with Embedded Coder

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

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

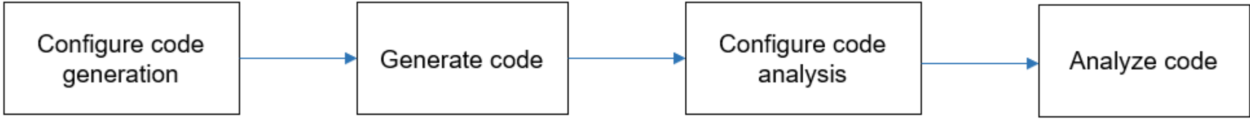
For a tutorial with a specific model, see “Run Polyspace Analysis on Code Generated from Simulink Model” on page 6-15.

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

Prerequisites

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

Generate and Analyze Code



Configure Code Generation and Generate Code

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

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

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

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

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

Configure Code Analysis

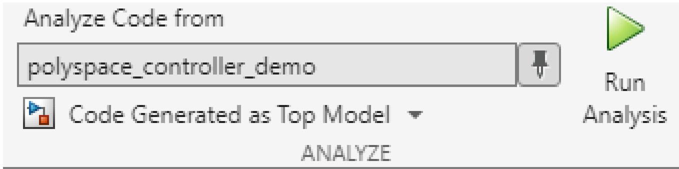
On the **Apps** tab, select **Polyspace Code Verifier**. On the **Polyspace** tab:

1. Select the product to run: **Bug Finder** or **Code Prover**. A Code Prover analysis detects run-time errors while a Bug Finder analysis detects coding defects and coding rule violations.
2. Select **Settings**. If needed, change default values of these options.

    - Settings from: Enable checking of MISRA™ coding rules in addition to the default checks specified in the project configuration. The default Bug Finder checks look for coding defects. The default Code Prover checks look for run-time errors.
    - “Input”, “Tunable parameters” and “Output”: Constrain inputs, tunable parameters, or outputs for a more precise Code Prover analysis.
    - “Output folder”: Specify a dedicated folder for results. The default analysis saves the results in a folder `results_`*`modelName`* in the current working folder.
    - “Open results automatically after verification”

Analyze Code

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



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

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

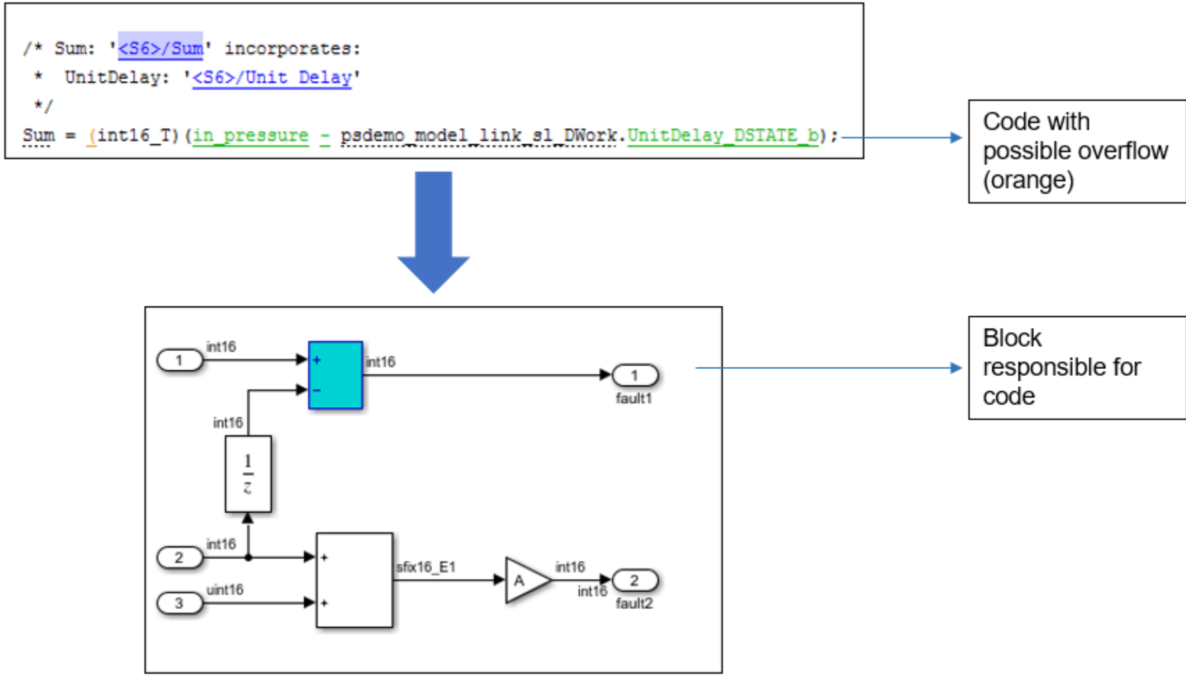
You can follow the progress of the analysis in the MATLAB Command Window.

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

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

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

Review Analysis Results



Review Results in Code

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

- “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2
- “Code Prover Result and Source Code Colors” on page 32-2
- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2
- “Filter and Group Results in Polyspace Desktop User Interface” on page 24-2

Navigate from Code to Model

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

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

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

Fix Issue

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

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

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

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

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

Annotate Blocks to Justify Issues

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

See Also

More About

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

Address Polyspace Results by Annotating Simulink Blocks

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

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

After you annotate a block, code operations generated from the block show results that are 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. Update your annotations when you update your model or generated code.

Annotate Blocks Through Polyspace User Interface

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

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

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

Source

psdemo_model_link_sl.c

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

- Print Source Code: psdemo_model_link_sl.c
- Search For "in_rotation" in Current Source File Ctrl+F
- Search For "in_rotation" in All Source Files
- Select Results At This Location (Ctrl+Click)
- Search For All References
- Go To Definition
- Go To Line Ctrl+L
- Go To Model
- Annotate Block
- Show Call
- Show In Variable Access View
- Open Editor
- Add Pre-Justification To Clipboard
- Expand All Macros
- Collapse All Macros
- Create Duplicate Code Window

Annotates the corresponding block in Model

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

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

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

Annotate Blocks in Simulink Editor

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

- Select the type of Polyspace result that you want to annotate from the drop-down list `Annotation Type`.
- If you want to annotate multiple results of the same type, enter a comma-separated list of result acronyms in the text box. See:
  - “Short Names of Bug Finder Defect Groups and Defect Checkers”
  - “Short Names of Code Prover Run-Time Checks” on page 30-12
- If you want to annotate only one result, select **Only 1 check**. The text box is converted into a dropdown list. Select the result that you want to annotate from this dropdown list.
- In the corresponding text boxes, enter the status, severity, and comment that you want to assign to the results.

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

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

Limitations

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

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

See Also

More About

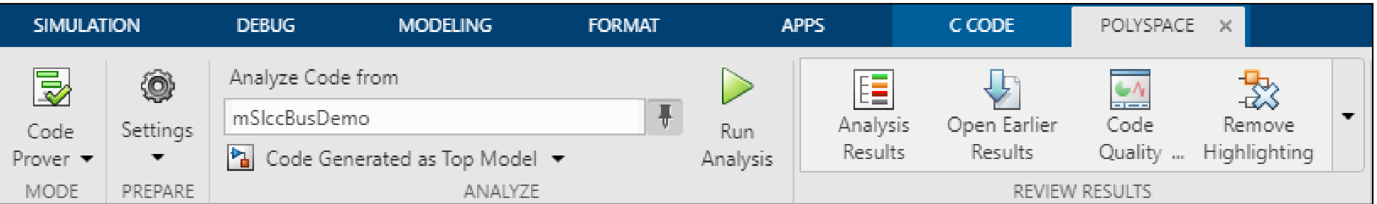
- “Configure Polyspace Options in Simulink” on page 6-53

Changes in Polyspace Analysis Workflows in Simulink in R2019b

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

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

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



Code Verification Workflow in a Nutshell

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

1 Select product to run:

For instance, select **Bug Finder**.

2 Specify code analysis options:

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

3 Specify which code to analyze:

Select whether to analyze the code generated for standalone use (typically, in the `*modelname*_ert_rtw` folder), the code generated for referencing in another context (typically, in the `slprj` 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 `*modelname*_ert_rtw` 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 `slprj` 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).	Select **Open Job Monitor**. 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”.	In the **Review Results** group, select **Remote Job Monitor**. 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”.
Open Polyspace Metrics or Polyspace Access web interface if you are using one of them to host Polyspace results. **Note** Polyspace Metrics is removed in R2021b and later releases.	Select **Open Metrics**. For opening a web interface, you must first specify the hostname and port number used for the web server in Polyspace preferences.	In the **Review Results** group, select **Code Quality Metrics** (Polyspace Metrics) or **Access** (Polyspace Access). For opening a web interface, you must first specify the hostname and port number used for the web server in Polyspace preferences.

See Also

More About

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

Run Polyspace on Code Generated by Using Previous Releases of Simulink

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

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

Prerequisite

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

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

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

Run a Cross-Release Polyspace Analysis

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

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

   ```
   % Load the model
   model = 'rtwdemo_roll';
   load_system(model);
   % Configure the Solver
   configSet = getActiveConfigSet(model);
   set_param(configSet,'Solver','FixedStepDiscrete');
   set_param(configSet, 'SystemTargetFile', 'ert.tlc');
   ```
2. The cross-release analysis requires packaging the generated code into a code archive. Set the option `PackageGeneratedCodeAndArtifacts` to `true`.

   ```
   set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true)
   ```
3. Create temporary folders for code generation and generate code.

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

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

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

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

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

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

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

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

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

   See “Complete List of Polyspace Code Prover Analysis Options”.

6. Start a Polyspace analysis.

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

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

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

Review Results

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

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

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

   You can upload the results to Polyspace Access. See “Upload Results to Polyspace Access” on page 2-28.

2. Review the results, and fix or justify the identified issues. For more information, see “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2.

See Also

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

More About

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

Run Polyspace Analysis on Code Generated from Simulink Model

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

Prerequisites

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

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

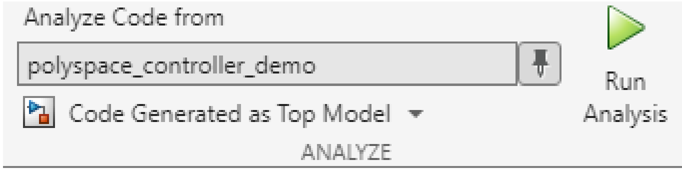
```
openExample('polyspace_code_prover/OpenSimulinkModelForPolyspaceAnalysisExample')
```

Open Simulink Model for Polyspace Analysis

```
Open the model polyspace_controller_demo.
```

Check for Run-Time Errors in Generated Code

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



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

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

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

Review Analysis Results

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

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

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

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

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

Trace and Fix Issues in the Model

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

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

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

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

Illegally dereferenced pointer

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

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

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

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

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

Out of bounds array index

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

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

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

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

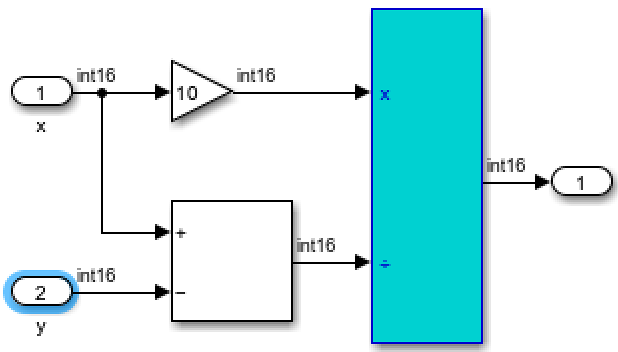
One solution for this error is to modify the prevailing conditions on `rtb_x` so that the expression `[(rtb_x - i) + 9]` evaluates to `0` or `1`.

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

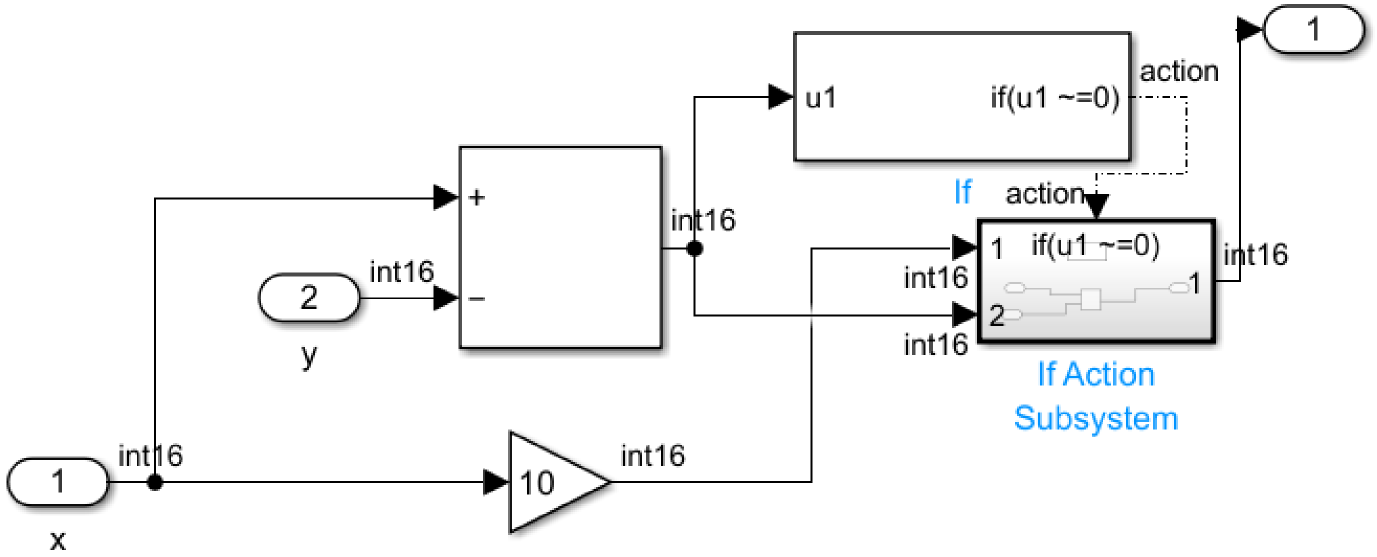
Orange checks

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

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

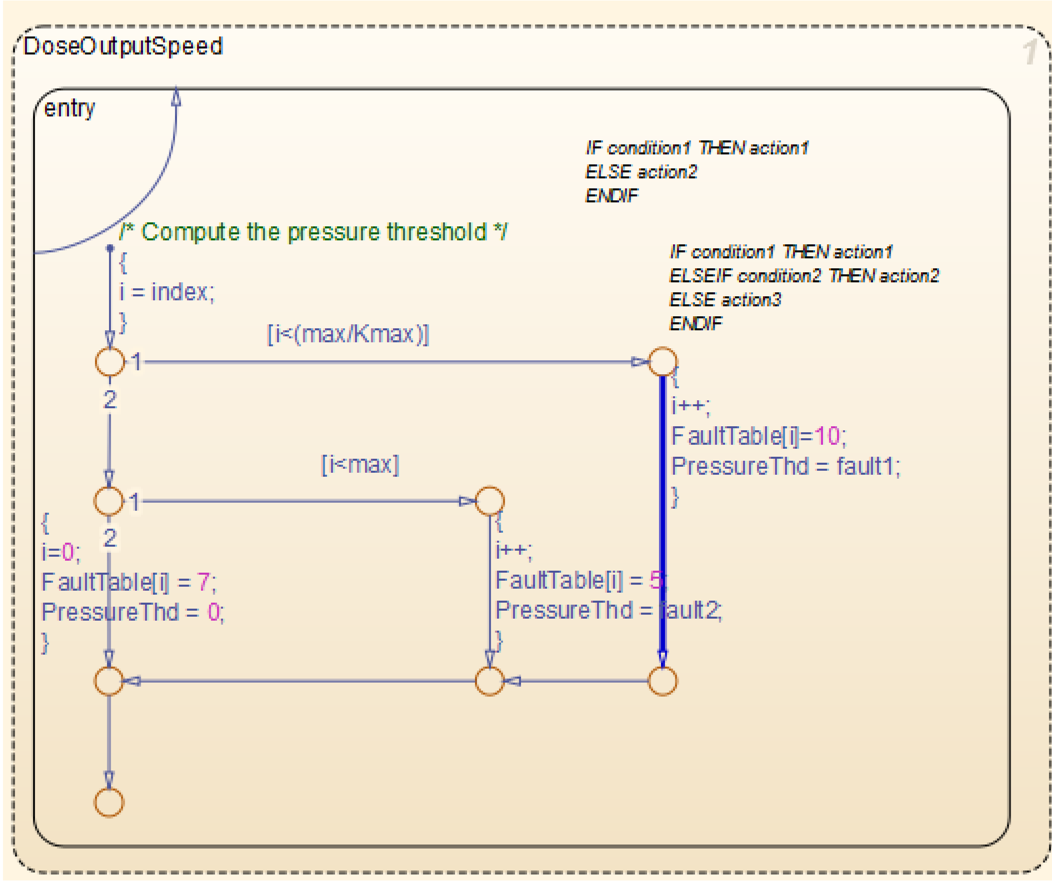
  

  To resolve this error, check that the ÷ input is not zero. For instance, use the If block and put the Divide block in an If Action Subsystem.



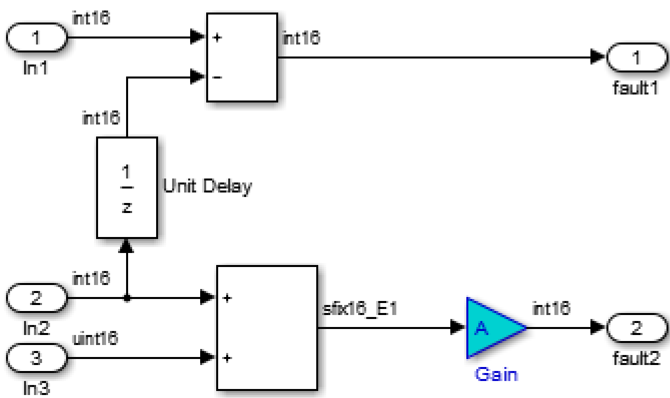
The other **Division by zero** checks can be resolved using similar techniques to check for a zero denominator.

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



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

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



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

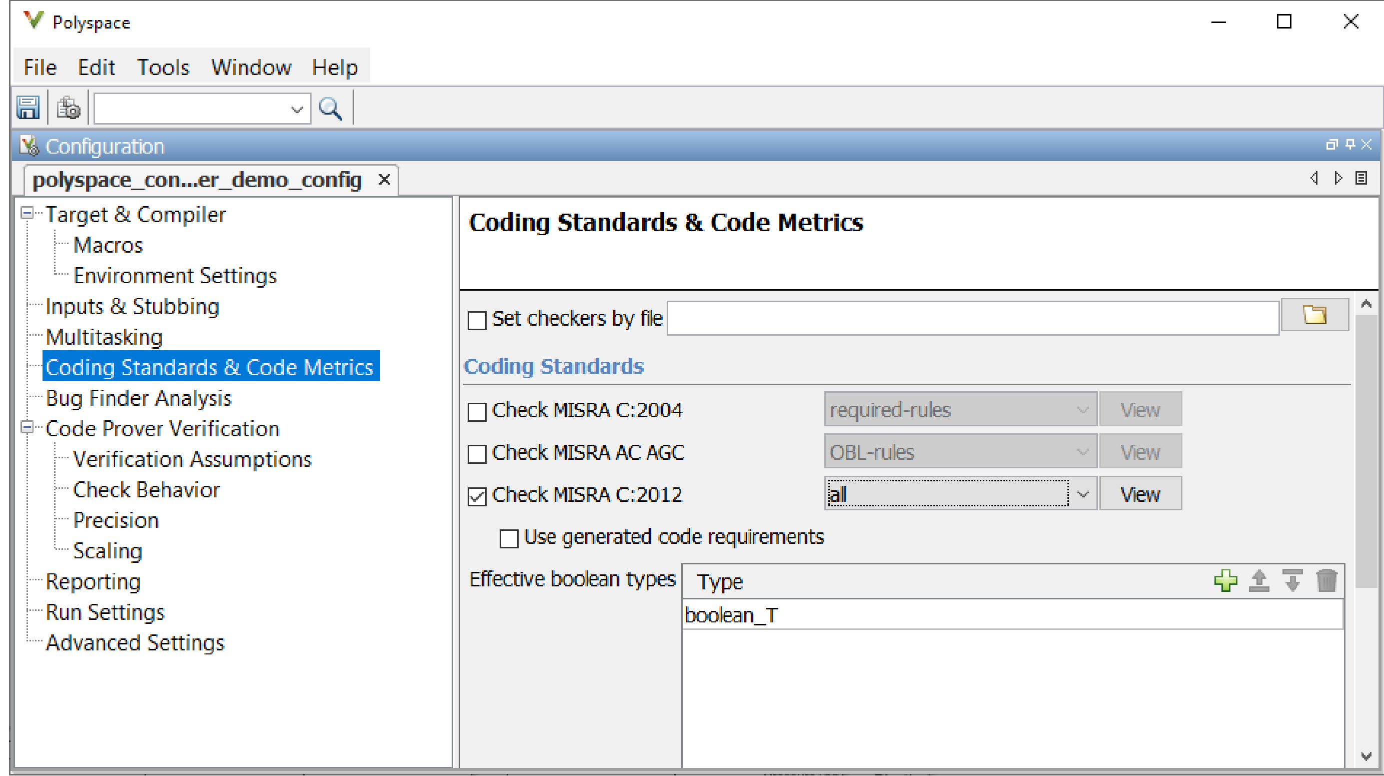
1. Double-click the Inport block `Battery info` that provides the input signal `in_battery_info` to the model.
2. On the **Signal Attributes** tab, change the **Maximum** value of the signal to a lower value, such as `500`.

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

Check for Coding Rule Violations

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

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



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

Alternatively, in the MATLAB Command Window, enter:

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

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

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

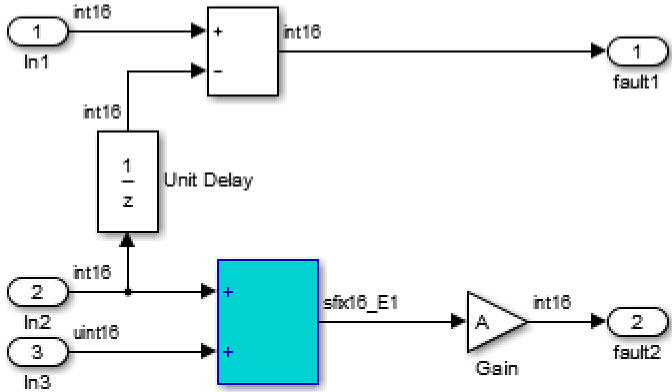
Annotate Blocks to Justify Results

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

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

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

   The rule violation occurs in a Sum block.



4. To annotate this block and justify the rule violation:

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

See Also

More About

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

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

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

Prerequisites

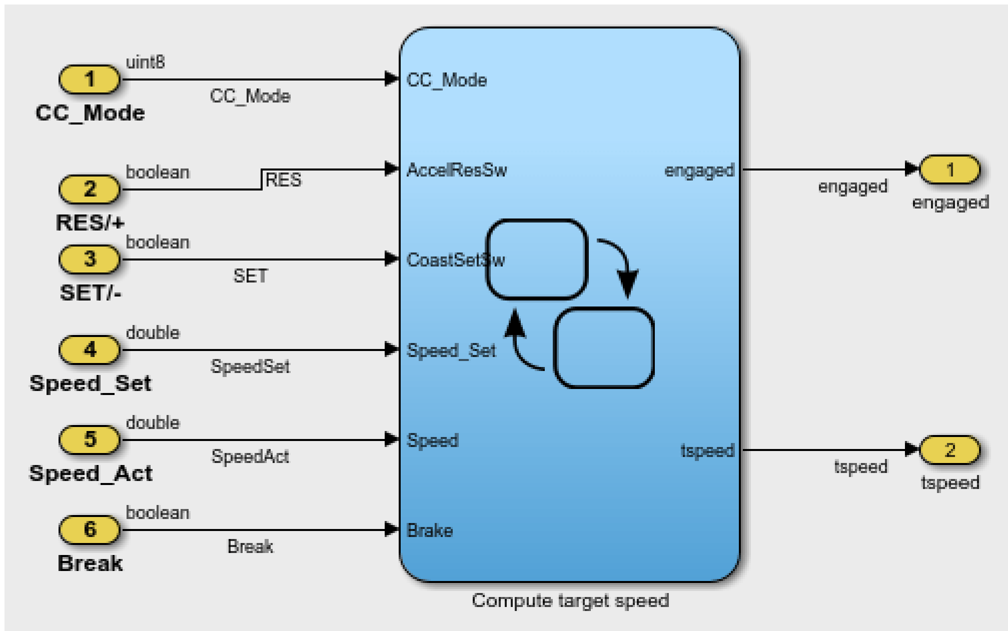
Before you run Polyspace from Simulink, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

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

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

Open Model

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



Detect and Fix Run-Time Errors

Detect Run-Time Errors

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

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

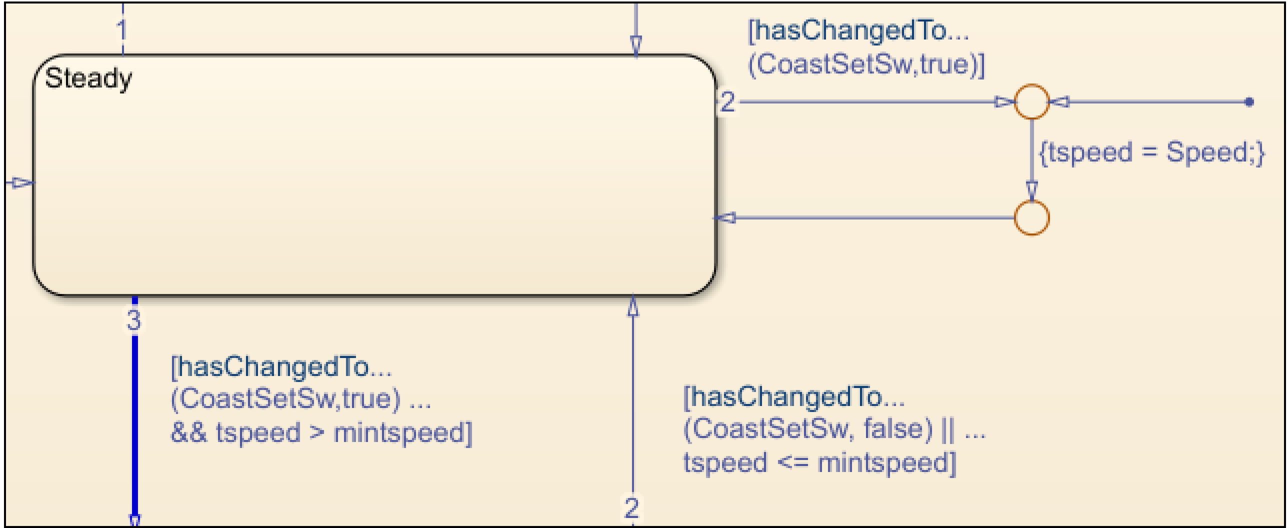
After the analysis is complete, the Code Prover results open in the Polyspace user interface. The results contain gray checks (unreachable code) and orange checks (potential run-time errors).

Fix Gray Checks

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

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

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

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



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

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

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

Fix Orange Checks

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

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

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

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

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

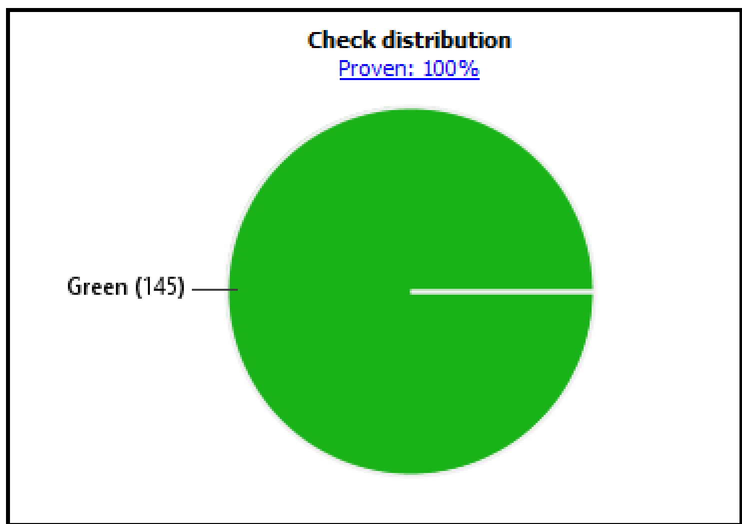
- Modify code:

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

  ```
  #define holdrate 5
  ```
- Modify analysis configuration:

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

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



Detect and Fix Coding Rule Violations

Detect MISRA C:2012 Violations

To detect MISRA C violations:

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

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

Fix MISRA C:2012 Violations

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

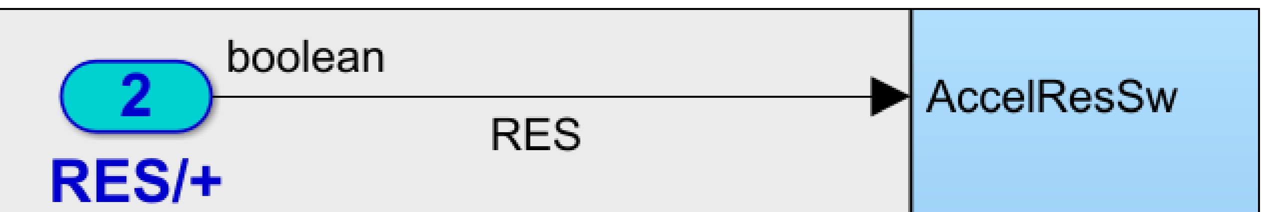
```
The character sequences /* and // shall not be used within a comment.
```

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

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

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

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



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

See Also

Related Examples

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

Run Polyspace Analysis on Generated Code by Using Packaged Options Files

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

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

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

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

Generate and Package Polyspace Options Files

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

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

   Alternatively, in the MATLAB Command Window, enter:

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

- To generate an archive of standalone generated code from the top model, use the function `slbuild`.
- To generate code as a model reference, use the function `slbuild`. After generating code as model reference, create the code archive by using the function `packNGo`.
- Alternatively, you can use TargetLink to generate the code. Create the code archive by archiving the generated code into a zip file.

5. To generate and package the Polyspace option files, in the MATLAB Command Window ,use the `polyspacePackNGo` function :

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

   See "Generate and Package Polyspace Options Files".

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

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

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

   See "Package Polyspace Options Files That Have Specific Polyspace Analysis Options".

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

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

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

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

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

Run Polyspace Analysis by Using the Packaged Options Files

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

1. Unzip the code archive and locate the `polyspace` folder.
2. On a Windows or Linux command line, run: *`productname`* `-options-file optionsFile.txt -results-dir` *`resultdir`*.

   - *`productname`* corresponds to one of: polyspace-bug-finder, polyspace-code-prover, polyspace-bug-finder-server, or polyspace-code-prover-server.
   - *`resultdir`* corresponds to the location of the Polyspace results. This argument is optional.

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

   If you are using Polyspace Access to view the results, upload the file `linksData.xml` in the same folder as your Polyspace results. You cannot link the code with Simulink model if you do not have the file `linksData.xml` or if you upload it outside the Polyspace result folder.
3. To review the result, upload it to Polyspace Access and view the results in a web browser. Alternatively, view the result by using the user interface of the Polyspace desktop products.

See Also

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

More About

- “Analyze Code Generated as Standalone Code in a Distributed Workflow” (Simulink)
- “Run Polyspace Analysis by Using MATLAB Scripts” on page 5-9
- “Integrate Polyspace Server Products with MATLAB” on page 10-33

Run Polyspace Analysis on Custom Code in Simulink Models

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

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

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

Prerequisite

Before you run Polyspace with Simulink, link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2 or “Integrate Polyspace Server Products with MATLAB” on page 10-33.

Analyze Custom Code

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

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

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

Specify Configuration

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

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

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

Start Polyspace Analysis

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

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

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

After extracting the code and model context, Polyspace analyzes them as handwritten code. See “Code Prover 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 Code Prover Results in Polyspace Desktop User Interface” on page 21-2
- “Code Prover Result and Source Code Colors” on page 32-2
- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2
- “Filter and Group Results in Polyspace Desktop User Interface” on page 24-2

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

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

See Also
`pslinkoptions` | `pslinkrun`

More About

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

Run Polyspace Analysis on S-Function Code

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

Prerequisites

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

S-Function Analysis Workflow

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

1. Compile your S-function to be compatible with Polyspace.
2. Select your Polyspace options.
3. Run a Polyspace Bug Finder analysis or a Polyspace Code Prover verification using one of the two analysis modes:
   - This Occurrence — Analyzes the specified occurrence of the S-function with the input for that block.
   - All Occurrences — Analyzes the S-function with input values from every occurrence of the S-function.
4. Review results in the Polyspace interface.
   - For information about navigating through your results, see “Filter and Group Results in Polyspace Desktop User Interface” on page 24-2.
   - For help reviewing and understanding the results, see “Complete List of Polyspace Code Prover Results”.

Compile S-Functions to Be Compatible with Polyspace

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

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

Example S-Function Analysis

This example shows the workflow for analyzing S-functions with Polyspace. You use the model `psdemo_model_link_sl` and the S-function `Command_Strategy`.

1. Open the model and use the Legacy Code Tool to compile the S-function `Command_Strategy`.

   ```
   % Open Model
   psdemo_model_link_sl
   ```

```
% Compile S-function Command_Strategy
def = legacy_code('initialize');
def.SourceFiles = { 'command_strategy_file.c' };
def.HeaderFiles = { 'command_strategy_file.h' };
def.SFunctionName = 'Command_Strategy';
def.OutputFcnSpec = 'int16 y1 = command_strategy(uint16 u1, uint16 u2)';
def.IncPaths = { fullfile(polyspaceroot, ...
  'toolbox','polyspace','pslink','pslinkdemos','psdemo_model_link_sl') };
def.SrcPaths = def.IncPaths;
def.Options.supportCoverageAndDesignVerifier = true;
legacy_code('compile',def);
```

2. Open the model `psdemo_model_link_sl/controller`.
3. Specify the code analysis options. On the **Apps** tab, select **Polyspace Code Verifier**. Then, on the **Polyspace** tab:

   - Select the product to run: **Bug Finder** or **Code Prover**. A Code Prover analysis detects run-time errors while a Bug Finder analysis detects coding defects and coding rule violations.
   - Select **Settings**. In the Configuration Parameters dialog box, make sure that the following parameters are set:

     - **Settings from** — Select `Project configuration`. Other values in the drop down menu enables checking different coding rules, which require using **Bug Finder** as the **Mode**.
     - **Open results automatically after verification** — ☑ On

   Apply your settings and close the Configuration Parameters.
4. Right-click the Command_Strategy block and select **Polyspace > Verify S-Function > This Occurrence**.
5. Follow the analysis in the MATLAB Command Window. When the analysis is finished, your results open in the Polyspace interface.

See Also

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 6-2
- “Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts” on page 6-37

Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts

You can check for bugs and run-time errors in the custom C/C++ code used in your Simulink model. The Polyspace analysis checks functions called from C Caller blocks and Stateflow charts with inputs from the model.

Prerequisites

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

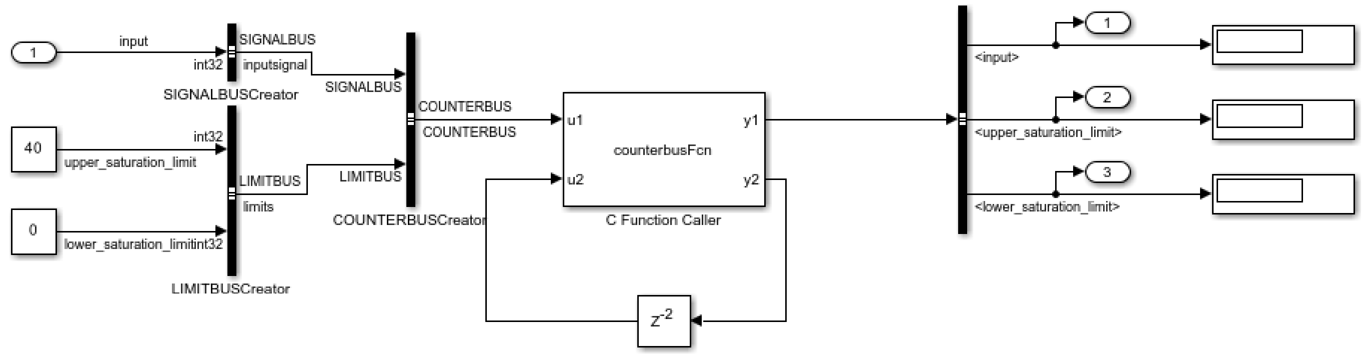
To open the models used in this example, look for this example in the MATLAB Help browser and click the **Open Model** buttons.

C/C++ Function Called Once in Model

This example uses a function called only once in the model from a C Caller block. The analysis checks the function using inputs to the C Caller block.

Open Model for Running Analysis on Custom Code

Open the model `mSlccBusDemo` for analyzing custom code with Polyspace. The model contains a C Caller block that calls a function `counterbusFcn` defined in a file `hCounterBus.c` (declared in file `hCounterBus.h`). The model uses variables saved in a MAT file `dLctData.mat`, which is loaded in the model using a callback.



Copyright 2018 The MathWorks, Inc.

Run Analysis

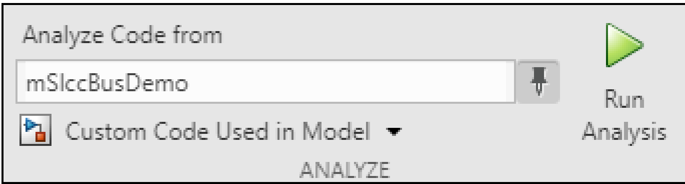
Configure analysis options and run Polyspace.

1. On the **Apps** tab, select **Polyspace Code Verifier** to open the **Polyspace** tab.
2. Specify the type of analysis:

- Select the product to run, **Bug Finder** or **Code Prover**. A Code Prover analysis detects run-time errors while a Bug Finder analysis detects coding defects and coding rule violations.
- Specify that the analysis must run on custom code in the model instead of generated code.

  The **Analyze Code from** field shows the model name. Below the field, instead of **Code Generated as Top Model**, select **Custom Code Used in Model**.

3 Select **Run Analysis**.



Follow the progress of analysis in the MATLAB Command Window. After the analysis, on the **Polyspace** tab, select **Analysis Results**. The results open in the Polyspace user interface.

You can also run the same analysis from MATLAB as follows. The script includes commands to load the model and the `.mat` file containing variables used in the model.

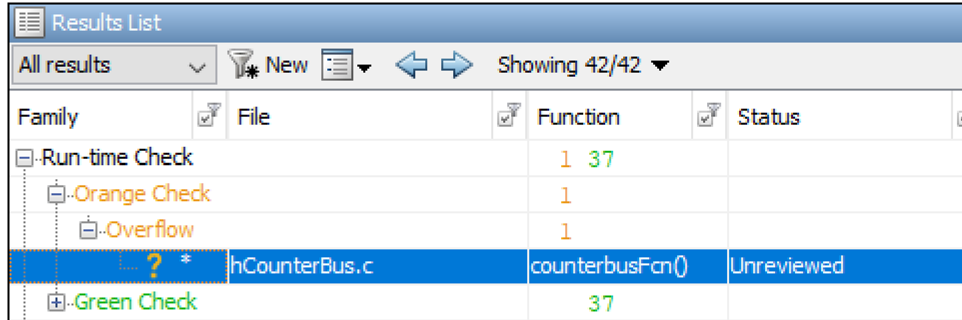
```
openExample('polyspace_code_prover/OpenModelForRunningAnalysisOnCustomCodeExample');
load_system('mSlccBusDemo');
load('dLctData.mat');

mlopts = pslinkoptions('mSlccBusDemo');
mlopts.VerificationMode = 'CodeProver';
pslinkrun('-slcc','mSlccBusDemo',mlopts);
```

Fix Issues

The analysis results appear on the **Results List** pane in the Polyspace user interface. Select each result and see further details on the **Result Details** pane and the corresponding source code on the **Source** pane.

The rest of this tutorial shows how to investigate and fix issues found in a Code Prover analysis. Similar steps can be followed for issues found with Bug Finder.

If you run a Code Prover analysis, the results contain an orange **Overflow** check.



The check highlights an addition operation in the `counterbusFcn` function that can overflow:

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

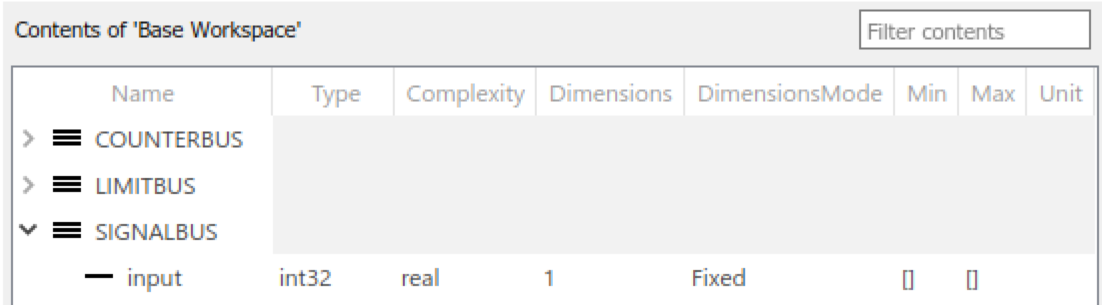
The operands come from inputs to `counterbusFcn`, which in turn come from these inputs to the C Caller block:

- The bus `COUNTERBUS`, which combines the signals `input`, `upper_saturation_limit`, and `lower_saturation_limit`. The signal `input` is unbounded.
- The feedback from the C Caller block itself through a Delay block.

You can constrain the signal named `input` in several ways. For instance, you can constrain the `Simulink.Bus` object named `SIGNALBUS` that contains `input`:

1. In the Simulink Toolstrip, on the **Modeling** tab, in the **Design** gallery, click **Type Editor**.

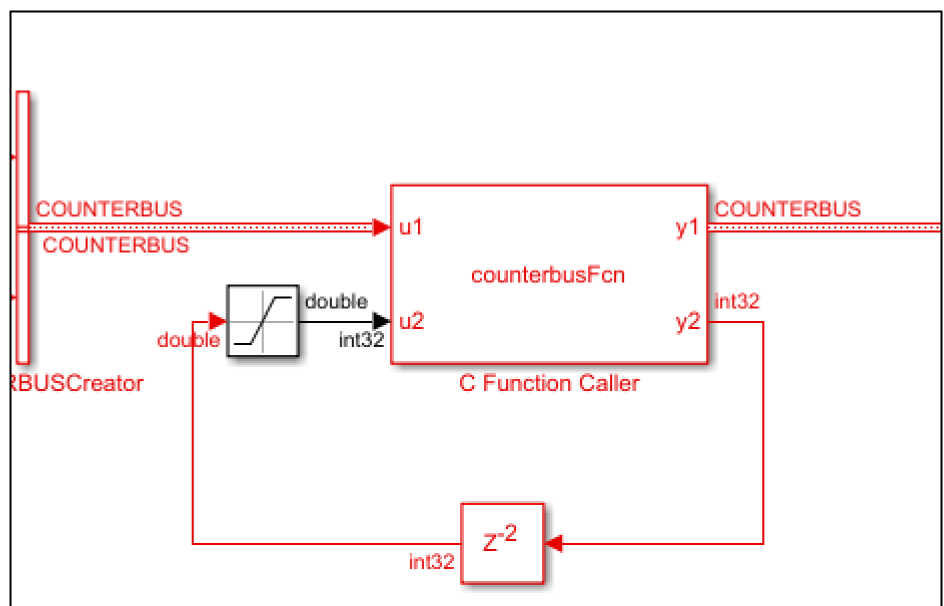
   The base workspace contains a `Simulink.Bus` object named `SIGNALBUS`.



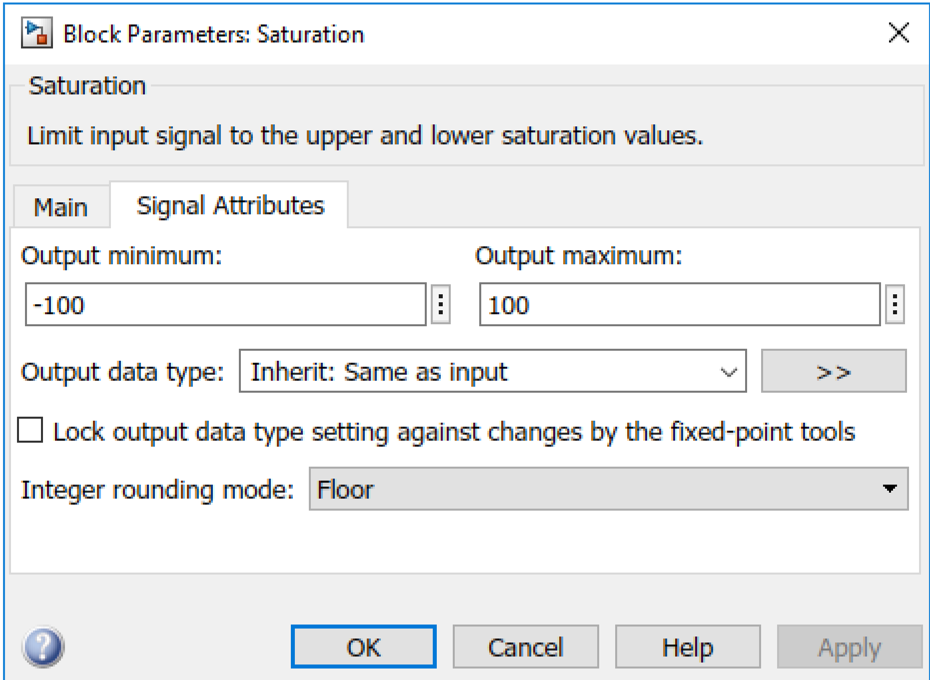
2. Specify a minimum and maximum value for the `input` element of `SIGNALBUS`.
3. Save the bus object in a MAT file. You can overwrite the file `dLctData.mat` or create a file.

You can also constrain the feedback from the C Caller block in several ways. For instance, you can saturate the feedback signal:

1. Add a Saturation block immediately before the feedback signal is input to the C Caller block.



2. On the **Signal Attributes** tab, specify a minimum and maximum value for the Saturation block output.



Note that specifying a lower and upper limit on the **Main** tab of the Saturation block is not sufficient to constrain the signal for the Polyspace analysis. The analysis uses the design ranges specified on the **Signal Attributes** tab.

Rerun the analysis. The **Overflow** check in the new set of results is green.

C/C++ Function Called Multiple Times in Model

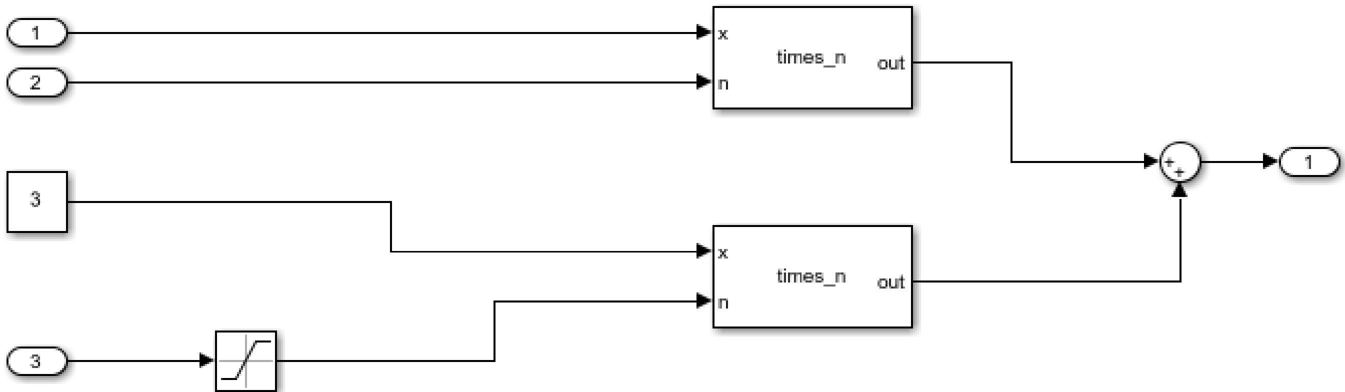
This example uses a function called from multiple C Caller blocks in the model. The function simply returns the product of its two arguments.

The example runs a Code Prover analysis and shows how to determine the function call context starting from Code Prover results. Typically, in a Bug Finder analysis, you do not need to distinguish between different call contexts.

Open Model for Analyzing All Custom Code

Open the model `multiCCallerBlocks` for running Polyspace analysis.

```
openExample('polyspace_bf/OpenModelForAnalyzingAllCustomCodeExample');
open_system('multiCCallerBlocks');
```



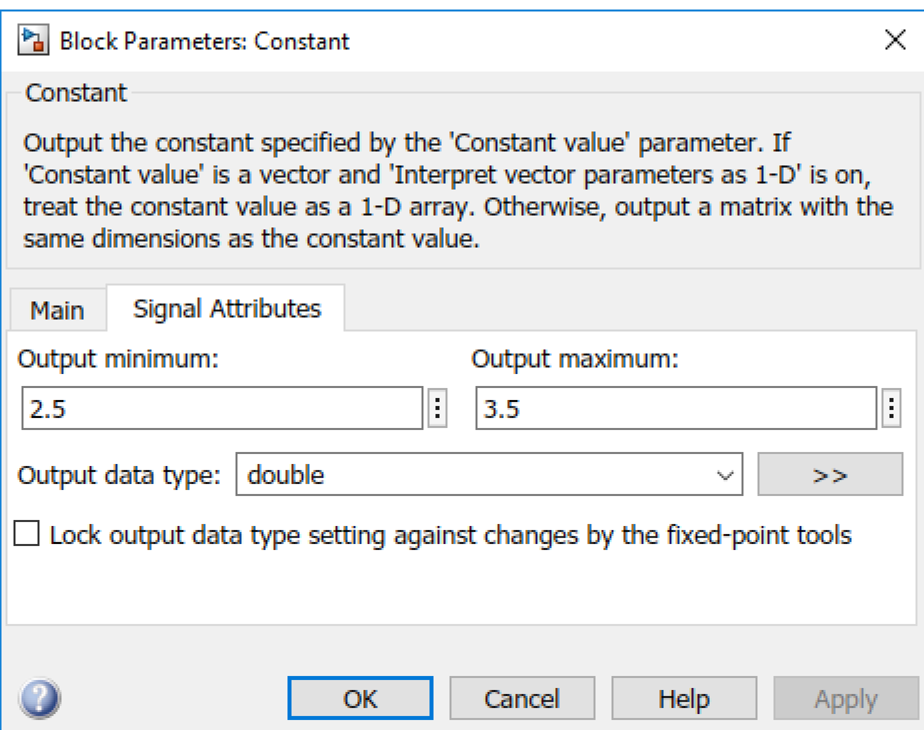
Inspect Model

The model contains two C Caller blocks calling the same function `times_n`. The inputs to one C Caller block come from two Inport blocks that have unbounded input. The inputs to the other C Caller block come from a Constant block and an Inport block that has the input bounded by a Saturation block.

To see the design ranges for the C Caller block that has bounded inputs:

- Double-click the Constant block or the Saturation block.
- On the **Signal Attributes** tab, note the design range.

  For instance, although the Constant block has the constant value set to 3, the design range for verification is 2.5 to 3.5.



The design range for the **Saturation** block is [-1,1].

Run Analysis and Review Results

Run analysis as in the previous example and open the results.

The **Results List** pane shows an orange **Overflow** check. The product in the `times_n` function overflows.

```
#include "file.h"

double times_n(double x, double n) {
  return x * n;
}
```

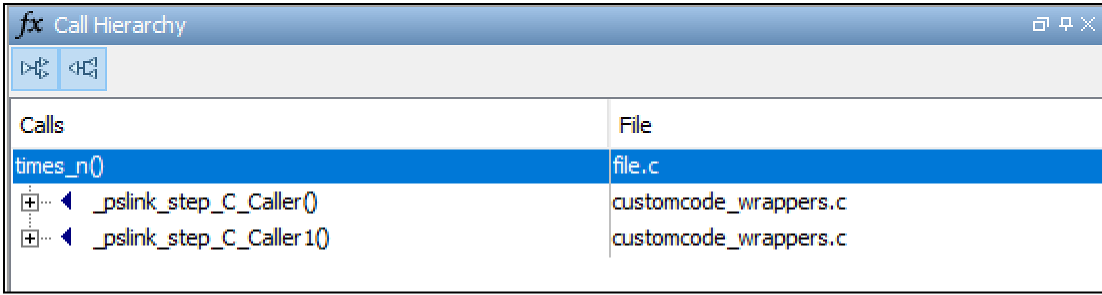
Because the `times_n` function is called from two contexts, the orange color combines both contexts and might indicate two possible situations:

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

To determine which call context leads to the overflow:

1. See all the callers of `times_n`.

   Select the orange **Overflow** check. On the **Result Details** pane, click . The **Call Hierarchy** pane shows the callers of `times_n`.
2. On the **Call Hierarchy** pane, you see two wrapper functions as callers. Each wrapper function represents a C Caller block in the model.

   Select one of the wrapper functions to open the source code for `customcode_wrappers.c`.
3. On the **Source** pane, inspect the code for the wrapper functions. To determine which inputs lead to the overflow, use tooltips on underlined inputs.

   For instance, the wrapper function for the C Caller block that has bounded inputs looks similar to this code:

```
/* Go to model '<Root>/C Caller1' */
/* Variables corresponding to inputs for block C Caller1 */
 real64_T _pslink_C_Caller1_In1;
 real64_T _pslink_C_Caller1_In2;
/* Variables corresponding to outputs for block C Caller1 */
```

```
 real64_T _pslink_C_Caller1_Out1;
/* Wrapper functions for code in block C Caller1 */
void _pslink_step_C_Caller1(void) {
    /* See tooltips on function inputs for input ranges */
    _pslink_C_Caller1_Out1 = times_n(_pslink_C_Caller1_In1, _pslink_C_Caller1_In2);
}
```

Use tooltips on the variables to determine their ranges. For instance, the tooltip on the variable `_pslink_C_Caller1_In1` shows that it is in the range [2.5, 3.5] and the tooltip on `_pslink_C_Caller1_In2` shows that it is in the range [-1,1]. Therefore, the product of the two inputs cannot overflow. The overflow must come from the other call context. You can see the tooltips on the inputs to the other call and confirm that the variables are unbounded.

To locate the C Caller block corresponding to a wrapper function, on the **Source** pane, click the blue block name link above the wrapper function (on the line starting with `Go to model`). The C Caller block is highlighted in the model.

Enable Context Sensitivity and Rerun Analysis

In this example, the function is simple enough that you can determine which call context leads to the overflow from the function inputs themselves. For more complex functions, you can configure the analysis to show results from the two contexts separately.

Because distinguishing call contexts involves a deeper analysis, the analysis might take longer. Therefore, enable context sensitivity only for specific functions and only if you are not able to distinguish the call contexts by inspection.

In this example, to enable context sensitivity for the `times_n` function:

1. In your model, on the **Polyspace** tab, select **Settings** > **Project Settings**.

   Alternatively, in the Polyspace user interface, select the **Project Browser**. Open the configuration of the project created for the analysis.
2. On the **Code Prover Verification** > **Precision** node, select `custom` for the option **Sensitivity context**. In the **Procedure** field, click [+] and enter `times_n`.

   See also `Sensitivity context (-context-sensitivity)`.

Rerun the analysis from the model and reopen the results. Select the orange **Overflow** check.

The **Result Details** pane shows the call contexts separately. You can see that the overflow occurs only for the call with unbounded inputs (row with orange text) and does not occur for the other call (row with green text).

Click the row with orange text to directly navigate to the wrapper function leading to the orange check. From the wrapper function, you can navigate to the C Caller block with unbounded inputs.

? **Overflow** (?)
Warning: operation [*] on float may overflow (on MIN or MAX bounds of FLOAT64)

Calling context	File	Scope	Line
operator * on type float 64 left: full-range [$-1.7977E^{+308}$ .. $1.7977E^{+308}$] right: full-range [$-1.7977E^{+308}$ .. $1.7977E^{+308}$]	customcode_wrappers.c	_pslink_step_C_Caller	26
operator * on type float 64 left: [2.5 .. 3.5] right: [-1.0 .. 1.0] result: [-3.5 .. 3.5]	customcode_wrappers.c	_pslink_step_C_Caller1	38

See Also

`pslinkoptions` | `pslinkrun`

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 6-2
- “Run Polyspace Analysis on S-Function Code” on page 6-35

Run Polyspace Analysis on Custom Code in C Function Block

You can run a Polyspace analysis on the custom C code in a C Function block from Simulink. Polyspace checks the custom C code for errors and bugs while keeping the model specific information such as design range specification, nature and number of inputs that are specified in the Simulink model.

Prerequisites

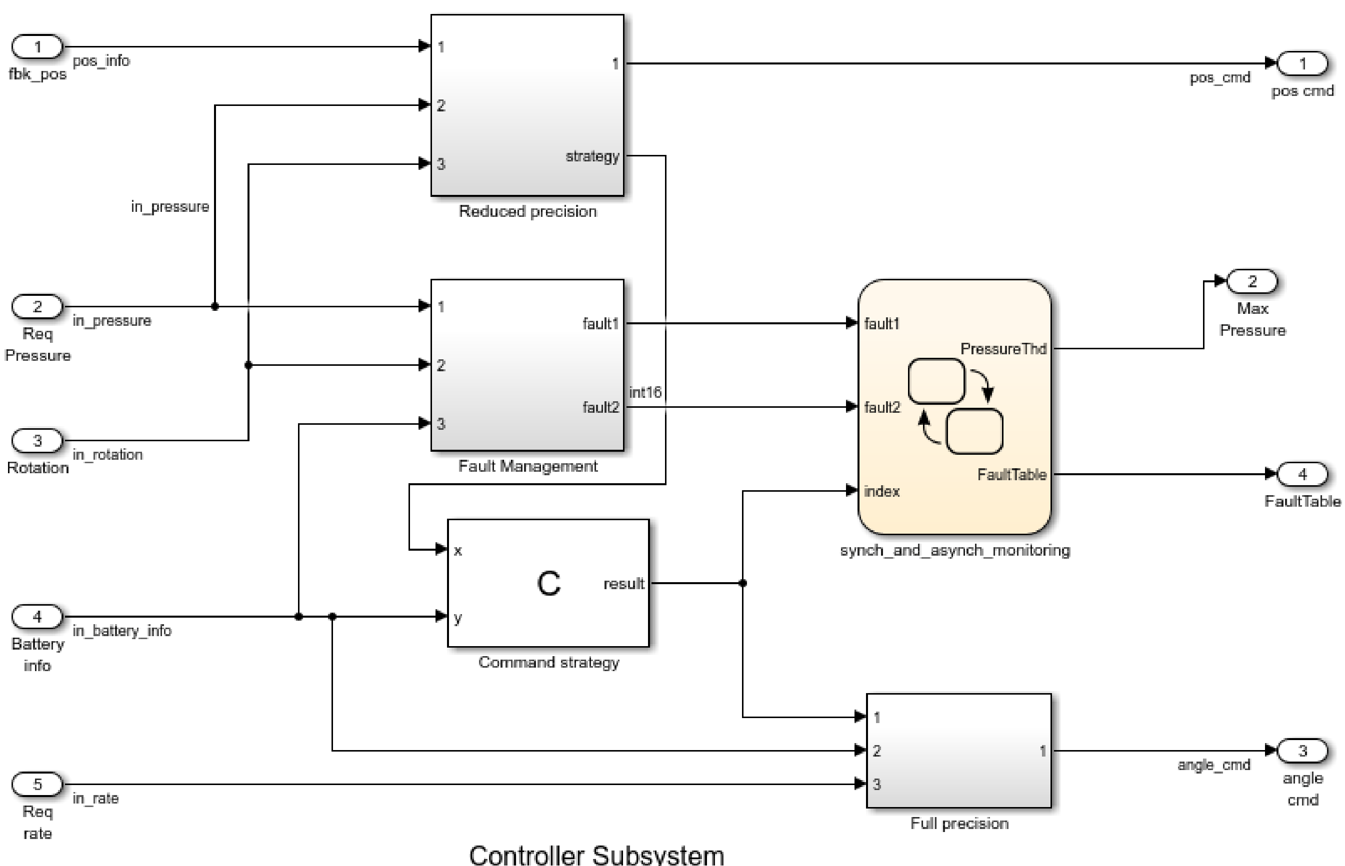
Before you run Polyspace from Simulink, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

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

```
openExample('polyspace_code_prover/CScriptDemoExample')
open_system('psdemo_model_link_sl_cscript');
```

Open Model for Running Polyspace Analysis on Custom Code in C Function Block

The model contains a C Function block called `Command Strategy` inside the `controller` subsystem.



Controller Subsystem

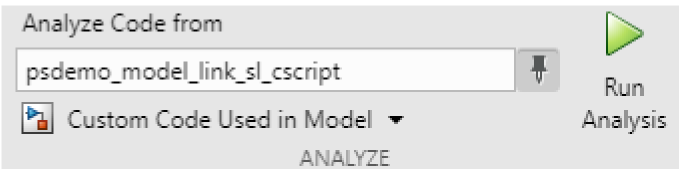
The `Command Strategy` block implements a look-up table using custom C code and outputs a value `result` based on two inputs `x` and `y`.

Run Polyspace Analysis

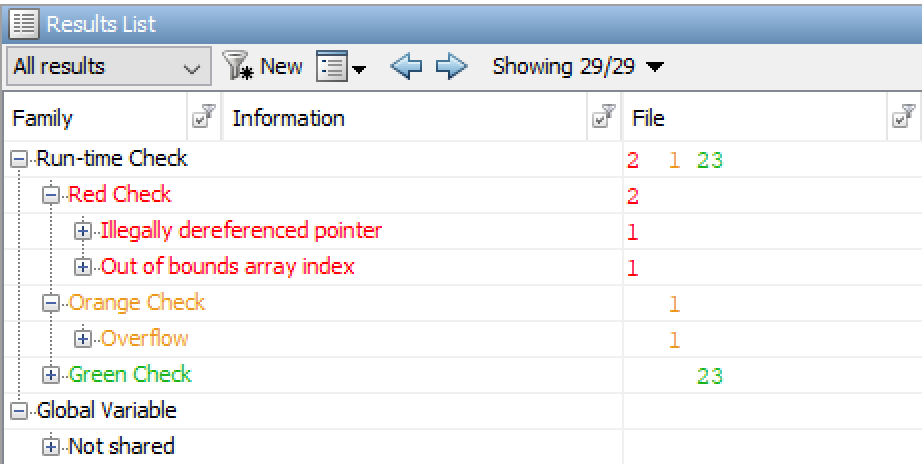
Run Polyspace Analysis from Simulink Editor

Click the **Apps** tab and select **Polyspace Code Verifier** to open the **Polyspace** tab.

1. Select **Bug Finder** or **Code Prover** in the **Mode** section of the **Polyspace** tab. A **Code Prover** analysis detects run-time errors while a **Bug Finder** analysis detects coding defects and coding rule violations.
2. To run a Polyspace analysis on the custom C code in the C Function block, select **Custom Code Used in Model** from the drop-down list in the **Analyze** section.



3. To start the Polyspace analysis, click the **Run Analysis** button. The MATLAB Command Window displays the progress of the analysis.
4. After the analysis, the Polyspace user interface opens with the results. You can choose to not open the results automatically after the analysis by unselecting **Open results automatically after verification** in **Settings**. To open the results after the analysis is finished, click the **Analysis Results** button.
5. To see all results of the Polyspace analysis, click **Clear active filters** from the **Showing** drop-down list in the **Results List** pane. If you run a **Code Prover** analysis, the results for the controller subsystem contain two red checks and an orange check.
6. To organize the results by family, click and select **Family**.



To switch between a **Bug Finder** and **Code Prover** analysis, return to the Simulink Editor from the Polyspace user interface. Switch between **Bug Finder** and **Code Prover** in the **Mode** section and run the analysis again.

Run Polyspace Analysis from MATLAB

You can run a Polyspace Code Prover analysis on the custom code for this model from MATLAB Editor or the Command Window using this code:

```
% Load the model 'psdemo_model_link_sl_cscript'
load_system('psdemo_model_link_sl_cscript');
% Create a 'pslinkoptions' object
mlopts = pslinkoptions('psdemo_model_link_sl_cscript');
% Specify whether to run 'CodeProver' or 'BugFinder' Analysis
mlopts.VerificationMode = 'CodeProver';
% Specify custom code as analysis target and run the analysis
pslinkrun('-slcc','psdemo_model_link_sl_cscript',mlopts);
```

Identify Issues in C Code

To identify issues in the custom C code, use the information in the **Result Details** pane and the **Source** pane of the Polyspace user interface. If you do not see these panes, go to **Window > Show/Hide View** and select the missing pane. For details on the panes, see “Result Details in Polyspace Desktop User Interface” on page 21-21 and “Source Code in Polyspace Desktop User Interface” on page 21-15.

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 Code Prover analysis, see “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2.

Fix Identified Issues

Modify the custom C code or the model to fix the issues. You can fix a Polyspace check in several ways. The examples here illustrate the general workflow of fixing Polyspace checks.

Illegally dereferenced pointer

You can address this check in several ways. Modify the C code so that a nonexistent memory address is not accessed.

1. Return to the Simulink Editor and double-click on the C Function block to open the custom code.
2. Use the index operator on array to access a valid array index. You can access indices from 0 to 99 because array has 100 elements. Accessing indices beyond this range results in a run-time error in Simulink.

   ```
   // access any index between 0 to 99
   tmp = array[50] + 5;
   ```

   Alternatively, assign the address of a valid memory location to p before the dereferencing operation. For example, *p can point to the 51st element in array.

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

Out of Bounds array index

You can address this check in several ways. Modify the code so that the size of another_array[] remains larger than or equal to the index value return_val-i+9.

1. Return to the Simulink Editor and double-click on the C Function block to open the custom code.
2. Modify the prevailing condition on return_val so that the index value return_val-i+9 always evaluates to 0 or 1.

   ```
   if ((return_val > 91) && (return_val < 93))
   //...
   ```

Alternatively, declare `another_array` with size 19.

```
int another_array[19];
```

Overflow

You can address this check in several ways as well. Modify the C code or the model so that the range of the right side of the assignment operation remains equal to or larger than that of the left side.

1. Return to the Simulink Editor.
2. Saturate the input variables `x` and `y` in the model so that their difference can fit into a 16-bit integer. The workflow for fixing **Overflow** by using saturation blocks is described in "Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts" on page 6-37.

Alternatively, increase the size of `return_val` in the custom C code to accommodate `x-y`.

1. Return to the Simulink Editor and double-click on the C Function block to open the custom code.
2. Declare `return_val` as a 32-bit integer.

```
int32_T return_val;
```

For details about addressing Polyspace results, see "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" on page 23-2.

See Also

`pslinkoptions` | `pslinkrun`

More About

- "Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts" on page 6-37
- "Complete List of Polyspace Code Prover 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: `SystemTargetFile (Simulink Coder)` Value: An Embedded Coder Target Language Compiler (TLC) file. For example `ert.tlc` or `autosar.tlc`.	Location: **Code Generation** Name: **System target file** Value: Embedded Coder target file
	Name: `MatFileLogging (Simulink Coder)` Value: `'off'`	Location: **Code Generation** > **Interface** Name: **MAT-file logging** Value: ☐ Not selected
	Name: `GenerateReport (Simulink Coder)` Value: `'on'`	Location: **Code Generation** > **Report** Name: **Create code-generation report** Value: ☑ Selected
	Name: `IncludeHyperlinkInReport (Embedded Coder)` Value: `'on'`	Location: **Code Generation** > **Report** Name: **Code-to-model** Value: ☑ Selected
	Name: `GenerateSampleERTMain (Embedded Coder)` Value: `'off'`	Location: **Code Generation** > **Templates** Name: **Generate an example main program** Value: ☐ Not selected
	Name: `GenerateComments (Simulink Coder)` Value: `'on'`	Location: **Code Generation** > **Comments** Name: **Include comments** Value: ☑ 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: ☐ Not selected
	Name: `ZeroExternalMemoryAtStartup (Embedded Coder)` Value: `'off'`	Location: **Optimization** Name: **Remove root level I/O zero initialization** Value: ☑ Selected
Solver	Name: `SolverType (Simulink)` Value: `'Fixed-Step'`	Location: **Solver** Name: **Type** Value: `Fixed-step`
	Name: `Solver (Simulink)` Value: `'FixedStepDiscrete'`	Location: **Solver** Name: **Solver** Value: `discrete (no continuous states)`

Configure Polyspace Options in Simulink

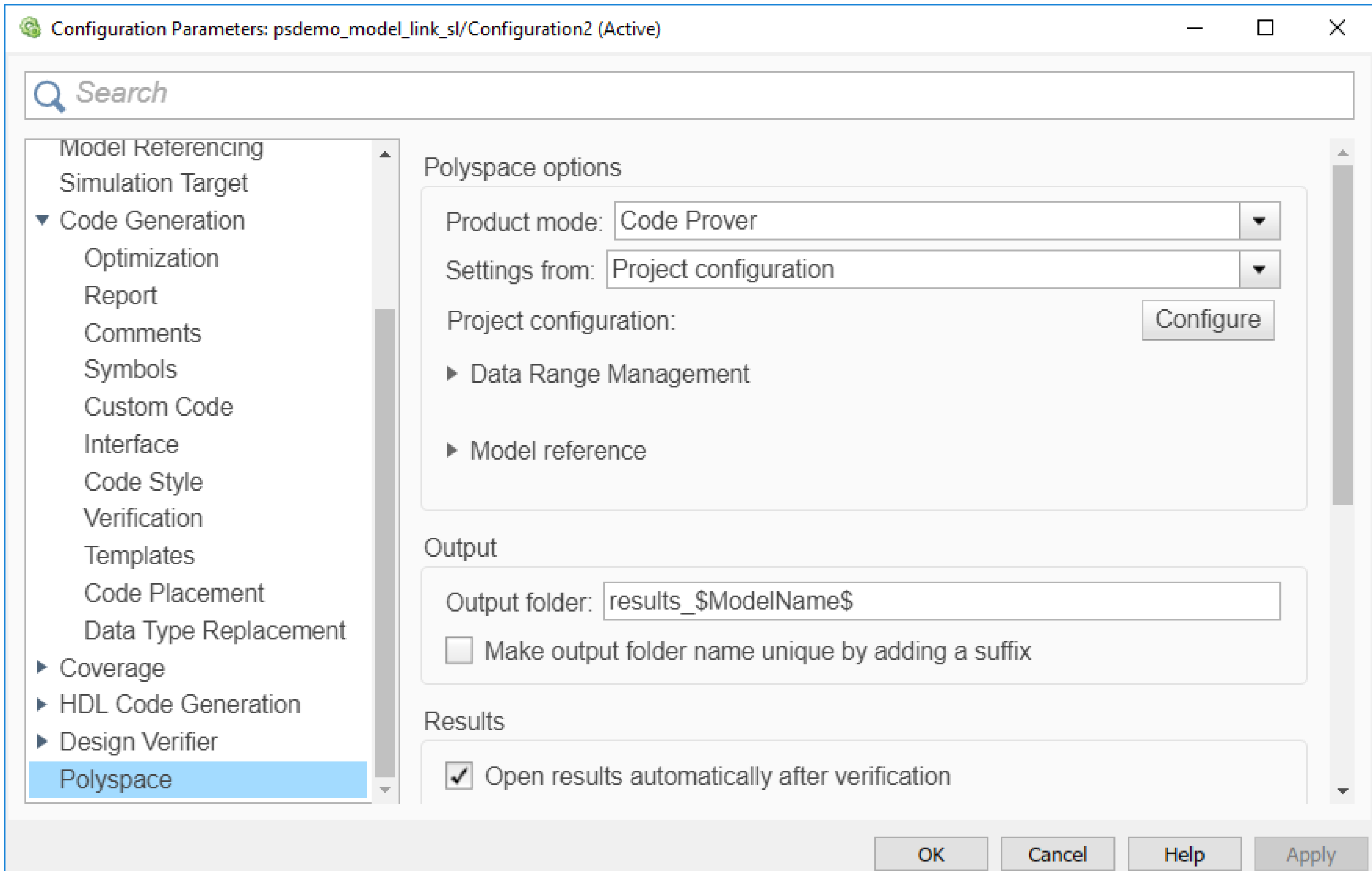
Configure basic and advanced Polyspace options when analyzing generated code. You can reuse existing configuration across multiple analysis.

To get started with Polyspace analysis in Simulink, see “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 6-2.

Configure Options

Set basic options

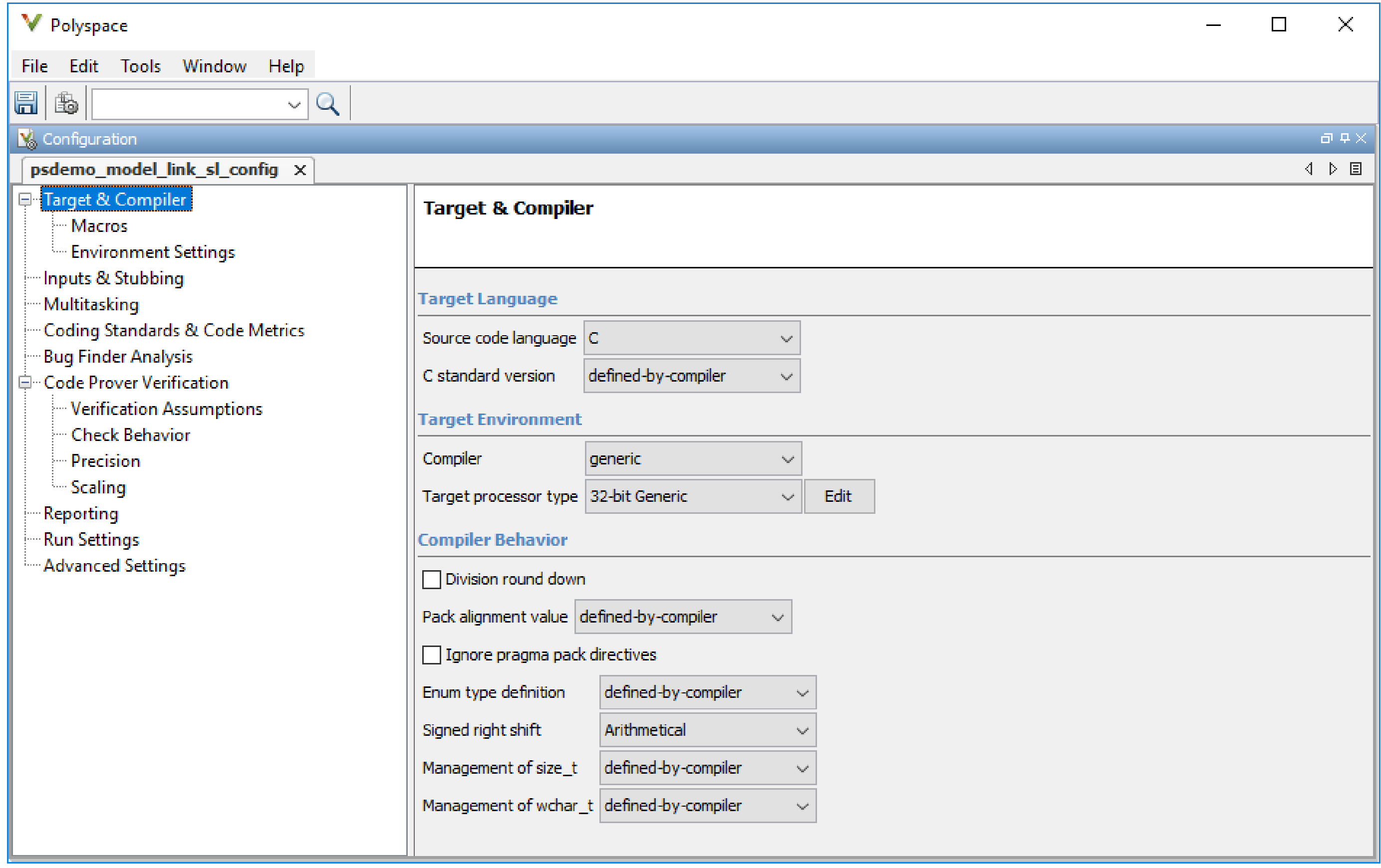
To set the basic Polyspace options in the Simulink Configuration Parameters window, on the **Apps** tab, select **Polyspace Code Verifier**. Then, on the **Polyspace** tab, select **Settings** or **Settings > Polyspace Settings**.



Set advanced options

The advanced options appear on the **Configuration** pane that also appears in the Polyspace user interface when you manually create a project for handwritten code.

To open the advanced options, on the **Polyspace** tab, select **Settings > Project Settings**.



On this pane, you can specify advanced settings.

- In the **Run Settings** pane, select options to run the code analysis on a remote cluster. Alternatively, in the **Advanced Settings** pane, use the option `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)` in the **Other** field.

  If you use this option, after starting the analysis, you can follow the analysis progress on the remote cluster through the Job Monitor window. On the **Polyspace** tab, select **Remote Job Monitor**.
- In the **Inputs & Stubbing** pane, specify options to stub certain functions for the analysis and then constrain the function output. Alternatively, in the **Advanced Settings** pane, use the options `Functions to stub (-functions-to-stub)` and `Constraint setup (-data-range-specifications)` in the **Other** field.

If a basic option in the Configuration Parameters window directly conflicts with an advanced option in the Polyspace window, the former prevails. For instance, say you specify these options:

- “Settings from (C)”: You select this basic option `Project configuration and MISRA C 2012 checking for generated code`.
- `Check MISRA C:2012 (-misra3)`: You disable this advanced option.

Polyspace ignores the advanced option and checks for violations of MISRA C:2012 rules.

By default, the advanced options are saved in the project file `*modelname*_config.psprj` in the `pslink_config` subfolder of the `results` folder. Use this project file to reuse the options associated with the project..

Share and Reuse Configuration

Share the basic or advanced options across multiple models.

- Basic options — Share and reuse the options set in the Configuration Parameters window. See “Share a Configuration with Multiple Models” (Simulink).
- Advanced options — Share and reuse the advanced options that are in a separate Polyspace project. Share this project across multiple models. When reusing advanced Polyspace options that are saved in a Polyspace project file, use a project file that is configured by using the **Polyspace App** in the Simulink Editor, as shown in “Set advanced options” on page 6-53. Reusing a project file that is not generated from the Simulink Editor can result in unexpected results.

You can specify the advanced options once, and then reuse the advanced options across multiple models. Set the basic options in each model individually.

Set options from model

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

Reuse options in another model

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

- Select **Use custom project file**. Provide the path to the `*.psprj` project file that you previously created.
- To use the project settings, select `Project configuration` under **Settings from**.

  If you want to check for additional issues, such as MISRA C: 2012 violations, select the options `Project configuration and MISRA C 2012 checking for generated code`.

If you run an analysis from the command line, you can set these options with the `pslinkoptions` function. See also pslinkoptions Properties.

See Also

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 6-2
- “Run Polyspace Analysis on Code Generated with TargetLink” on page 6-62
- “Default Polyspace Options for Code Generated with Embedded Coder” on page 6-57
- “Default Polyspace Options for Code Generated with TargetLink” on page 6-64

How Polyspace Analysis of Generated Code Works

When you generate code from a Simulink model, the generated code can contain these components:

- `initialize()` functions that run before the simulation starts.
- `terminate()` functions that run after the simulation ends.
- `step()` functions that run in a loop to perform the simulation.

Additionally, the generated code might have a placeholder `main()` function that contains calls to the above. You might edit the placeholder `main()` to fit your deployment purposes. For more information about the `main` generated by Embedded Coder, see "Main Program" (Embedded Coder).

When you run Polyspace on generated code, Polyspace gathers this information from your code:

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

When you run Code Prover, the software uses this information to generate a separate `main()` function to facilitate the analysis. Regardless of the presence of the generated placeholder `main()`, Polyspace uses its own `main()` function that performs these tasks:

1. Initializes parameters by using the Polyspace option `Parameters (-variables-written-before-loop)`.
2. Calls initialization functions by using the option `Initialization functions (-functions-called-before-loop)`.
3. Initializes inputs using the option `Inputs (-variables-written-in-loop)`.
4. Calls the `step` function in a loop by using the option `Step functions (-functions-called-in-loop)`. By default, Polyspace assumes that the `step` function might be called an arbitrary number of times in the loop. To specify the number of iterations in the loop for a more precise Code Prover analysis, use the option `-main-generator-bounded-loop`.
5. Calls the `terminate` function by using the option `Termination functions (-functions-called-after-loop)`.

The Polyspace generated `main` function might have this structure:

```
init parameters    \\ -variables-written-before-loop
init_fct()             \\ -functions-called-before-loop
  while(random){        \\ start main loop with one or more iterations
  init inputs      \\ -variables-written-in-loop
  step_fct()       \\ -functions-called-in-loop
}
terminate_fct()    \\ -functions-called-after-loop
```

For C++ code generated with Embedded Coder, the `initialize()`, `step()`, and `terminate()` functions and associated variables are either class members or have global scope.

Default Polyspace Options for Code Generated with Embedded Coder

In this section...

"Default Options" on page 6-57
"Constraint Specification" on page 6-57
"Recommended Polyspace options for Verifying Generated Code" on page 6-58
"Hardware Mapping Between Simulink and Polyspace" on page 6-58

Default Options

For Embedded Coder code, the software sets the following verification options by default:

```
-sources path_to_source_code
-D PST_ERRNO
-D main=main_rtwec
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
-functions-to-stub=[rtIsNaN,rtIsInf,rtIsNaNF,rtIsInfF]
-results-dir results
```

Note *matlabroot* is the MATLAB installation folder.

Constraint Specification

You can constrain inputs, parameters, and outputs to lie within specified ranges. Use these configuration parameters:

- "Input"
- "Tunable parameters"
- "Output"

The software automatically creates a Polyspace constraints file using information from the MATLAB workspace and block parameters.

You can also manually define a constraints file in the Polyspace user interface. See "Specify External Constraints for Polyspace Analysis" on page 14-2. If you define a constraints file, the software appends the automatically generated information to the constraints file you create. Manually defined constraint information overrides automatically generated information for all variables.

The software supports the automatic generation of constraint specifications for the following kinds of generated code:

- Code from standalone models
- Code from configured function prototypes

- Reusable code
- Code generated from referenced models and submodels

Additional Information

See also “External Constraints on Polyspace Analysis of Generated Code” on page 6-59.

Recommended Polyspace options for Verifying Generated Code

For Embedded Coder code, the software automatically specifies values for the following verification options:

- `-main-generator`
- `-functions-called-in-loop`
- `-functions-called-before-loop`
- `-functions-called-after-loop`
- `-variables-written-in-loop`
- `-variables-written-before-loop`

Embedded Coder performs a wraparound of the variables in the generated code that might overflow. When running a Code Prover analysis of code generated by Embedded Coder, Polyspace uses these options:

- `-signed-integer-overflows warn-with-wrap-around`
- `-unsigned-integer-overflows allow`

These options might have different default values when analyzing code that is not generated by Embedded Coder. See `Overflow mode for signed integer (-signed-integer-overflows)` and `Overflow mode for unsigned integer (-unsigned-integer-overflows)`.

In addition, for the option `-server`, the software uses the value specified in the **Send to Polyspace server** check box on the **Polyspace** pane. These values override the corresponding option values in the **Configuration** pane of the Polyspace user interface.

You can specify other verification options for your Polyspace Project through the Polyspace **Configuration** pane. See “Configure Polyspace Options in Simulink” on page 6-53.

Hardware Mapping Between Simulink and Polyspace

The software automatically imports target word lengths and byte ordering (endianness) from Simulink model hardware configuration settings. The software maps **Device vendor** and **Device type** settings on the Simulink **Configuration Parameters > Hardware Implementation** pane to **Target processor type** settings on the Polyspace **Configuration** pane.

The software creates a generic target for the verification.

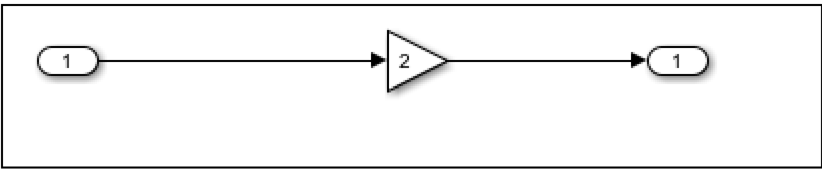
External Constraints on Polyspace Analysis of Generated Code

When you check generated code for bugs or run-time errors, you can choose whether to perform the check for all values of an input or a specific range of values. You can extract the input range from the Simulink model, or specify your own external constraints.

Likewise, you can use a fixed value for tunable parameters or a range of values. You can also check whether output values fall within a specific range.

Extract External Constraints from Model

Consider this simple model with an Inport block, a Gain block, and an Outport block. Suppose the signal in the Inport and Outport blocks and the gain parameter of the Gain block have a minimum and maximum value.



You can analyze the code generated from this model with these minimum and maximum values. On the **Apps** tab, select **Polyspace Code Verifier**. Then, on the **Polyspace** tab, select **Settings**. Specify these configuration parameters:

- “Input”: Select `Use specified minimum and maximum values`. The Code Prover analysis checks the generated code within the specified range of values from the Inport block. The Bug Finder analysis uses this information to exclude false positives.

  Default: This option is selected.
- “Tunable parameters”: Select `Use specified minimum and maximum values`.

  Default: This option is not selected. The analysis uses the fixed gain value of the Gain block (the value 2 in the example).

  For the analysis to consider a range instead of a fixed value, the parameters must be tunable and not inlined. See **Default parameter behavior**.
- “Output”: Select `Verify outputs are within minimum and maximum values`. The Code Prover analysis creates a red check if the outputs exceed the range specified on the Outport block. See also `Correctness condition`.

  Default: This option is not selected. The Code Prover analysis does not check output values.

After analysis, to check if a constrained range value is used, see one of these files:

- Constraint specification XML file *modelname*`_drs.xml` in the folder `results_`*modelname*`\`*modelname*.
- Polyspace project file *modelname*`.prpsj` in the folder `results_`*modelname*.

  Open this file in the Polyspace user interface. In the project configuration, see the extracted constraints specified for the option `Constraint setup (-data-range-specifications)`.

Storage Classes Supported for Constraint Extraction From Simulink Model

To allow constraint extraction from the Simulink model, the signals and parameters must have data types in specific storage classes. For details on storage classes, see “Choose Storage Class for Controlling Data Representation in Generated Code” (Embedded Coder).

Common Storage Classes

Storage Class	Signal Constraint Supported	Parameter Constraint Supported
`Auto`	Yes	Yes
`ExportedGlobal`	Yes	Yes
`ImportedExtern`	Yes	Yes
`ImportedExternPointer`	Yes	Yes
`Model default`	Yes	Yes

Other Storage Classes

Storage Class	Signal Constraint Supported	Parameter Constraint Supported
`BitField`	Yes	Yes
`CompilerFlag`	No	No
`Const`	No	Yes
`ConstVolatile`	No	Yes
`Define`	No	No
`ExportToFile`	Yes	Yes
`FileScope`	Yes	No
`GetSet`	No	No
`ImportedDefine`	No	No
`ImportFromFile`	No	No
`Struct`	No	No
`Volatile`	Yes	Yes

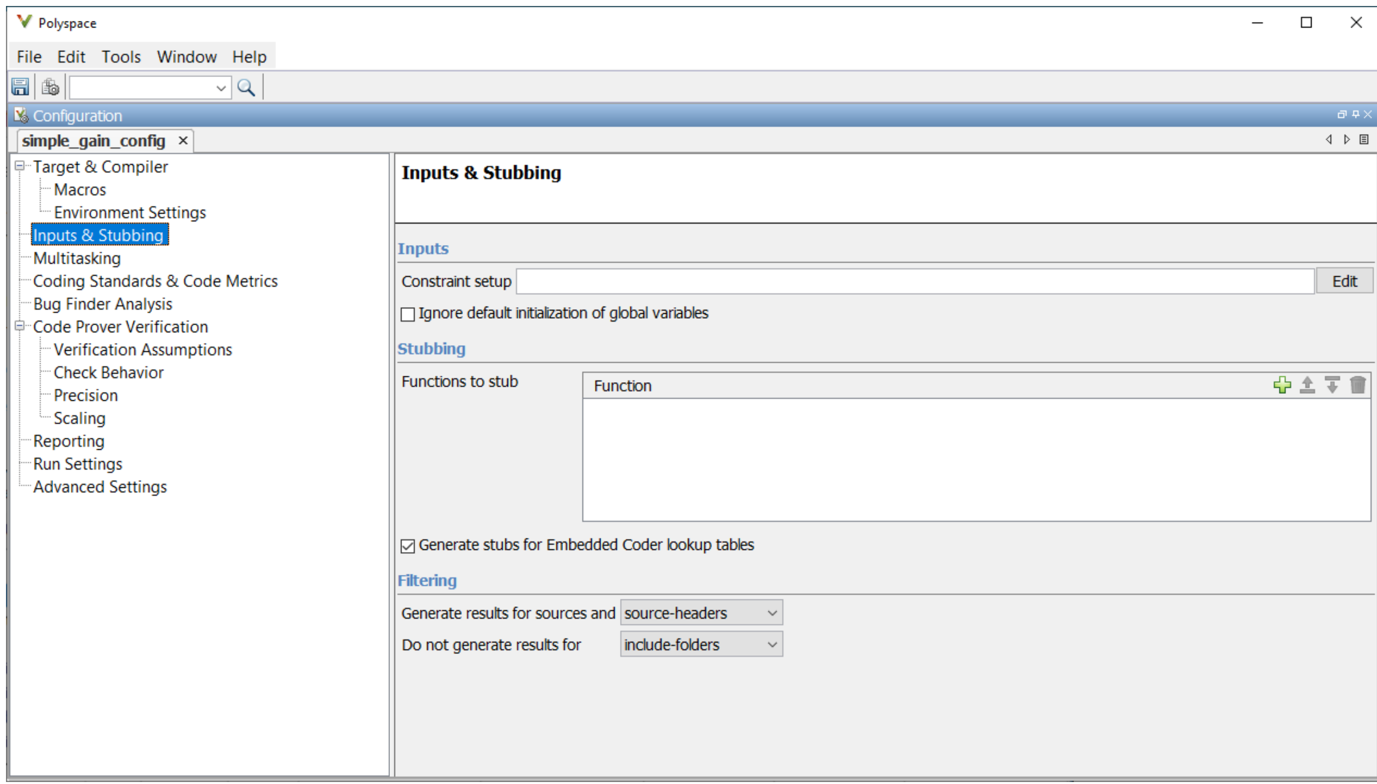
Specify Custom External Constraints

In some instances, you might need to specify a custom set of constraints on your generated code. For instance, you might be integrating the generated code with an existing code base, which imposes a set of custom constraints.

When analyzing the generated code, specify custom external constraints through the Polyspace Configuration window:

1. In the Simulink Configuration Parameters window, locate the **Polyspace** tab, and then click **Configure** to open the Polyspace Configuration window.

2 In the **Constraint Setup** field, located in the **Inputs & Stubbing** node, specify the custom external specification XML file.



You can create and edit a custom external constraint template through the Polyspace user interface. See "Specify External Constraints for Polyspace Analysis" on page 14-2.

See Also

More About

- "Default Polyspace Options for Code Generated with Embedded Coder" on page 6-57
- "Choose Storage Class for Controlling Data Representation in Generated Code" (Embedded Coder)
- "Specify External Constraints for Polyspace Analysis" on page 14-2
- "External Constraints for Polyspace Analysis" on page 14-6

Run Polyspace Analysis on Code Generated with TargetLink

To detect bugs and runtime errors, run a Polyspace analysis after generating code from Simulink models by using TargetLink. Run the analysis from the Simulink Editor window. Manually setting up a Polyspace project is not necessary. If you use Embedded Coder to generate code, see “Run Polyspace Analysis on Code Generated from Simulink Model” on page 6-15.

Configure and Run Analysis

Configure code analysis

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

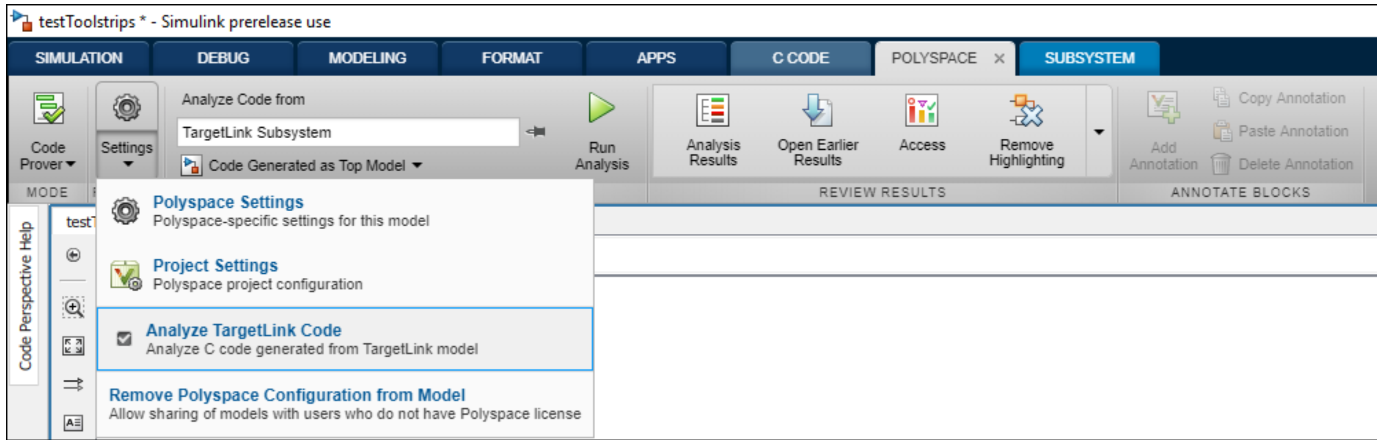
- Select the product to run: **Bug Finder** or **Code Prover**. A Code Prover analysis detects run-time errors while a Bug Finder analysis detects coding defects and coding rule violations.
- Select **Settings**. Change default values of these options if needed.
  - “Settings from (C)”: Enable checking of MISRA or JSF® coding rules in addition to the default checks.
  - “Output folder”: Specify a dedicated folder for results. The default analysis runs Code Prover on generated code and saves the results in a folder `results_`*`modelName`* in the current working folder.
  - “Enable additional file list”: Add C files that are not part of the generated code.
  - “Open results automatically after verification”

Analyze code

To analyze generated code:

1. Choose to analyze code generated from a TargetLink Subsystem. You cannot analyze code generated from the entire model.

   The **Analyze Code from** field shows the top model. Unpin the content of this field and then select the TargetLink Subsystem.
2. Select **Settings** > **Analyze TargetLink Code**. Then, select **Run Analysis**.



You can follow the progress of the analysis in the MATLAB command window.

The results open automatically unless explicitly disabled. By default, the results are saved in a folder `results_ModelName` in the current folder. Each new run overwrites previous results. You can change these behaviors or save the results to a Simulink project using appropriate configuration parameters.

Note Verification of a 3,000 block model takes approximately one hour to verify, or about 15 minutes per 2,000 lines of generated code.

Review Analysis Results

Review result in code

The results appear on the **Results List** pane. Click each result to see the source code and details on the **Result Details** pane.

Navigate from code to model

Links in code comments show blocks that generate the subsequent lines of code. To see the blocks in the model, click the block names.

Fix issue

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

For instance, you might need to constrain the range of signal from Inport blocks. See “Work with Signal Ranges in Blocks” (Simulink). If a flagged issue is known or justified, then annotate that information in the relevant blocks. To annotate a block in Simulink Editor, right-click the block and use the contextual menu.

Default Polyspace Options for Code Generated with TargetLink

In this section...

"TargetLink Support" on page 6-64
"Default Options" on page 6-64
"Lookup Tables" on page 6-64
"Data Range Specification" on page 6-65
"Code Generation Options" on page 6-65

TargetLink Support

The Windows version of Polyspace Code Prover is compatible with dSPACE® Data Dictionary and TargetLink Code Generator.

Polyspace Code Prover 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 Code Prover extracts information from the dSPACE Data Dictionary, you must regenerate the code before performing an analysis.

Default Options

Polyspace sets the following options by default:

```
-sources path_to_source_code
-results-dir results_folder_name
-I path_to_source_code
-D PST_ERRNO
-I dspaceroot\matlab\TL\SimFiles\Generic
-I dspaceroot\matlab\TL\srcfiles\Generic
-I dspaceroot\matlab\TL\srcfiles\i86\LCC
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
-functions-to-stub=[rtIsNaN,rtIsInf,rtIsNaNF,rtIsInfF]
-scalar-overflows-behavior wrap-around
-boolean-types Bool
```

Note *dspaceroot* and *matlabroot* are the dSPACE and MATLAB tool installation directories respectively.

Lookup Tables

By default, Polyspace provides stubs for the lookup table functions. The dSPACE data dictionary is used to define the range of their return values. A lookup table that uses extrapolation returns full range for the type of variable that it returns. You can disable this behavior from the Polyspace configuration menu.

Data Range Specification

You can constrain inputs, parameters, and outputs to lie within specified data ranges. See “Work with Signal Ranges in Blocks” (Simulink).

The software automatically creates a Polyspace constraints file using the dSPACE Data Dictionary for each global variable. The constraint information is used to initialize each global variable to the range of valid values as defined by the min..max information in the data dictionary. This information allows Polyspace software to model real values for the system during analysis. Carefully defining the min-max information in the model allows the analysis to be more precise, because only the range of real values is analyzed.

Note Boolean types are modeled having a minimum value of 0 and a maximum of 1.

You can also manually define a constraint file in the Polyspace user interface. See “Specify External Constraints for Polyspace Analysis” on page 14-2. If you define a constraint file, the software appends the automatically generated information to the constraint file you create. Manually defined constraint information overrides automatically generated information for all variables.

Constraints cannot be applied to static variables. Therefore, the compilation flags `-D static=` is set automatically. It has the effect of removing the static keyword from the code. If you have a problem with name clashes in the global name space, either rename the variables or disable this option in Polyspace configuration.

Code Generation Options

From the TargetLink Main Dialog, it is recommended to:

- Set the option `Clean code`
- Unset the option `Enable sections/pragmas/inline/ISR/user attributes`
- Turn off the compute to overflow (CTO) generation. Polyspace can analyze code generated with CTO, but the results may not be as precise.

When you install Polyspace, the `tlcgOptions` variable is updated with `'PolyspaceSupport', 'on'` (see variable in `'C:\dSPACE\Matlab\Tl\config\codegen\tl_pre_codegen_hook.m'` file).

See Also

Related Examples

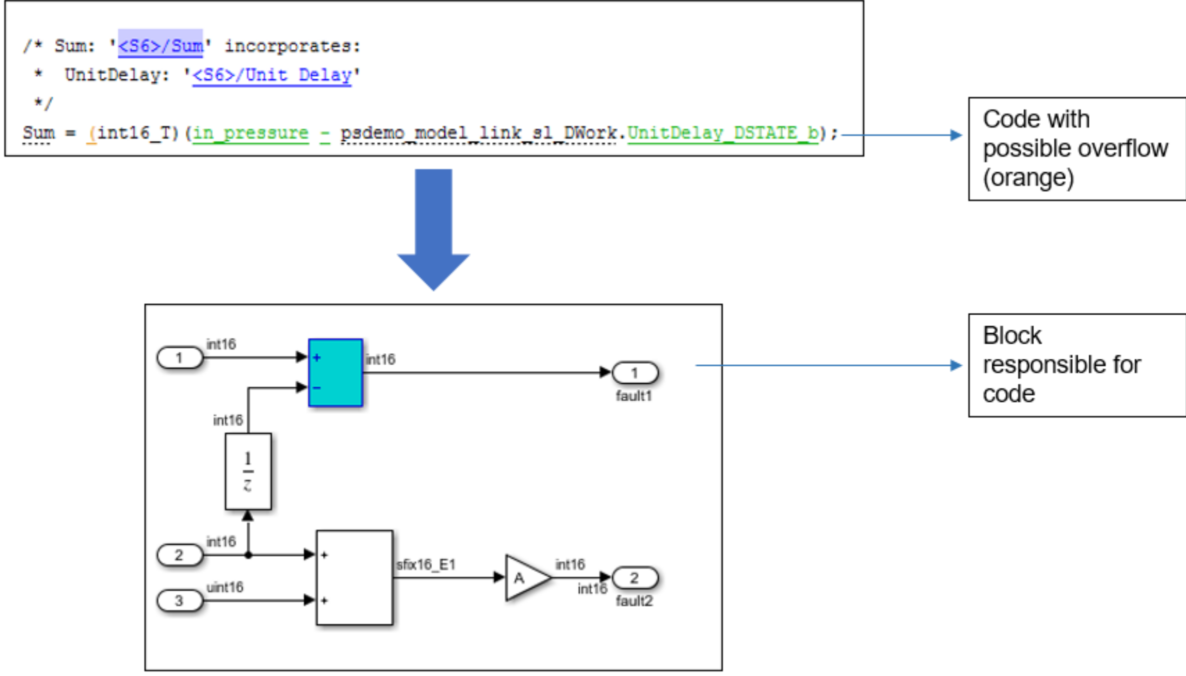
- “Run Polyspace Analysis on Code Generated with TargetLink”

External Websites

- dSPACE – TargetLink

Troubleshoot Navigation from Code to Model

When you run Polyspace on generated code, in the analysis results, you see links in code comments. The links show names of blocks that generate the subsequent lines of code. To see the blocks in the model, you click the block names in the links.



This topic shows the issues that can happen in navigation from code to model.

Links from Code to Model Do Not Appear

See if you are looking at source files (`.c` or `.cpp`) or header files. Header files are not directly associated with blocks in the model and do not have links back to the model.

Links from Code to Model Do Not Work

You may encounter issues with the back-to-model feature if:

- Your operating system is Windows Vista® or Windows 7; and User Account Control (UAC) is enabled or you do not have administrator privileges.
- You have multiple versions of MATLAB installed.

To reconnect MATLAB and Polyspace:

1. Close Polyspace.
2. At the MATLAB command-line, enter `pslinkfun('enablebacktomodel')`.

When you open your Polyspace results, the hyper-links will highlight the relevant blocks in your model.

Your Model Already Uses Highlighting

If your model extensively uses block coloring, the coloring from this feature may interfere with the colors already in your model. You can change the color of blocks when they are linked to Polyspace results. For instance, to change the color to magenta, use this command:

```
color = 'magenta';
HILITE_DATA = struct('HiliteType', 'find', 'ForegroundColor', 'black', ...
        'BackgroundColor', color);
set_param(0, 'HiliteAncestorsData', HILITE_DATA)
```

The color can be one of the following:

- 'cyan'
- 'magenta'
- 'orange'
- 'lightBlue'
- 'red'
- 'green'
- 'blue'
- 'darkGreen'

Polyspace Support of MATLAB and Simulink from Different Releases

Polyspace support of MATLAB or Simulink varies depending on their respective releases. Polyspace fully supports MATLAB and Simulink from the same release, offering complete integration with these software. Polyspace supports MATLAB and Simulink from earlier releases with cross-release integration. See the table.

	Polyspace Release R2018a	Polyspace Release R2018b	Polyspace Release R2019a	Polyspace Release R2019b	Polyspace Release R2020a	Polyspace Release R2020b	Polyspace Release R2021a	Polyspace Release R2021b	Polyspace Release R2022a	Polyspace Release R2022b	Polyspace Release R2023a
MATLAB or Simulink Release R2018a	**Complete Integration** on page 6-70	"Partial Integration" on page 6-71	"Partial Integration" on page 6-71	"Partial Integration" on page 6-71	"Partial Integration" on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71
MATLAB or Simulink Release R2018b	* on page 6-71	**Complete Integration** on page 6-70	"Partial Integration" on page 6-71	"Partial Integration" on page 6-71	"Partial Integration" on page 6-71	"Partial Integration" on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71
MATLAB or Simulink Release R2019a	* on page 6-71	* on page 6-71	**Complete Integration** on page 6-70	"Partial Integration" on page 6-71	"Partial Integration" on page 6-71	"Partial Integration" on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71
MATLAB or Simulink Release R2019b	* on page 6-71	* on page 6-71	* on page 6-71	**Complete Integration** on page 6-70	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71

MATLAB or Simulink Release R2020a	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	**Complete Integration** on page 6-70	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71
MATLAB or Simulink Release R2020b	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	**Complete Integration** on page 6-70	“Cross-Release Integration” on page 6-70	“Cross-Release Integration” on page 6-70	“Cross-Release Integration” on page 6-70	“Cross-Release Integration” on page 6-70	“Cross-Release Integration” on page 6-70
MATLAB or Simulink Release R2021a	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	**Complete Integration** on page 6-70	“Cross-Release Integration” on page 6-70	“Cross-Release Integration” on page 6-70	“Cross-Release Integration” on page 6-70	“Cross-Release Integration” on page 6-70
MATLAB or Simulink Release R2021b	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	**Complete Integration** on page 6-70	“Cross-Release Integration” on page 6-70	“Cross-Release Integration” on page 6-70	“Cross-Release Integration” on page 6-70
MATLAB or Simulink Release R2022a	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	**Complete Integration** on page 6-70	“Cross-Release Integration” on page 6-70	“Cross-Release Integration” on page 6-70

MATLAB or Simulink Release R2022b	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	**Complete Integration** on page 6-70	“Cross-Release Integration” on page 6-70
MATLAB or Simulink Release R2023a	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	* on page 6-71	**Complete Integration** on page 6-70

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

Complete Integration

If MATLAB and Polyspace are from the same release, you can integrate them after installation by calling `polyspacesetup`. See “Same Release of Polyspace and MATLAB” on page 5-2.

You can:

- Run a Polyspace analysis from the Simulink Editor or from the MATLAB Command Window on C/C++ code that is generated from a model or included as custom code in a model. Annotate Simulink blocks and navigate back-to-model from the Polyspace user interface.

  See “Code Prover 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 “Code Prover Analysis in MATLAB Coder”.
- Run a Polyspace analysis on handwritten C/C++ code by using MATLAB scripts.

  See “Code Prover Analysis with MATLAB Scripts”.

Cross-Release Integration

You can integrate Polyspace with MATLAB or Simulink from a release after R2020b. See “MATLAB Release Earlier Than Polyspace” on page 5-3.

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

- To run a Polyspace analysis on C/C++ code generated by using Embedded Coder, in the MATLAB Command Window, call these functions:

  - `pslinkrunCrossRelease`
  - `pslinkfun`
  - `pslinkoptions`
- Navigate back to your Simulink model from the Polyspace user interface.

You cannot:

- Start a Polyspace analysis of generated code from the Simulink Editor or MATLAB Coder App.
- Start a Polyspace analysis of the custom code included in models or handwritten C/C++ code in the MATLAB Command Window.
- Start a Polyspace analysis of C/C++code generated from MATLAB code in the MATLAB Command Window.

See “Run Polyspace on Code Generated by Using Previous Releases of Simulink” on page 6-12.

Partial Integration

You can partially integrate Polyspace with MATLAB or Simulink from a release earlier than R2020b. See “MATLAB Release Earlier Than Polyspace” on page 5-3.

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

- To run a Polyspace analysis on C/C++ code generated by using Embedded Coder, in the MATLAB Command Window, call these functions:
  - `pslinkrun`
  - `pslinkfun`
  - `pslinkoptions`
- Navigate back to your Simulink model from the Polyspace user interface.

You cannot:

- Start a Polyspace analysis of generated code from the Simulink Editor or MATLAB Coder App.
- Start a Polyspace analysis of the custom code included in models or handwritten C/C++ code in the MATLAB Command Window.
- Start a Polyspace analysis of C/C++code generated from MATLAB code in the MATLAB Command Window.

Navigate Back to Model

You can navigate back to your Simulink model from the Polyspace user interface without integrating Polyspace into your MATLAB or Simulink. Polyspace does not integrate with MATLAB and Simulink if:

- Your MATLAB or Simulink is from a more recent release than your Polyspace.
- Your MATLAB or Simulink is more than four releases behind your Polyspace.

Some specific releases of MATLAB or Simulink do not integrate with Polyspace. See the table.

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

- Identify the comments in your code that act as links to the Simulink model. In the **Tools > Preferences > Miscellaneous** tab, select your code generation tool from the context menu **Code comments that act as code-to-model links**. Polyspace recognizes Embedded Coder, MATLAB Coder, and TargetLink. If you use a different code generating tool, select **User Defined**. In the field **Comments beginning with**, specify prefixes of the code comments that act as links.
- In the **Source** pane of the Polyspace user interface, click the code comments that appear as hyperlinks.

See Also

`polyspacesetup` | `pslinkrunCrossRelease`

More About

- "Integrate Polyspace with MATLAB and Simulink" on page 5-2
- "Run Polyspace on Code Generated by Using Previous Releases of Simulink" on page 6-12
- "Fix Issues When when Integrating Polyspace with MATLAB and Simulink" on page 34-73

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, *`polyspacroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2023a`. The example shows a Code Prover analysis. You can follow a similar workflow for Bug Finder.

Prerequisites

To run this tutorial:

- You must have an Embedded Coder license. The MATLAB Coder app does not show options for running Polyspace unless you have an Embedded Coder license.
- You must be familiar with how to open and use the MATLAB Coder app or the `codegen` command. Otherwise, see the MATLAB Coder Getting Started.
- You must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

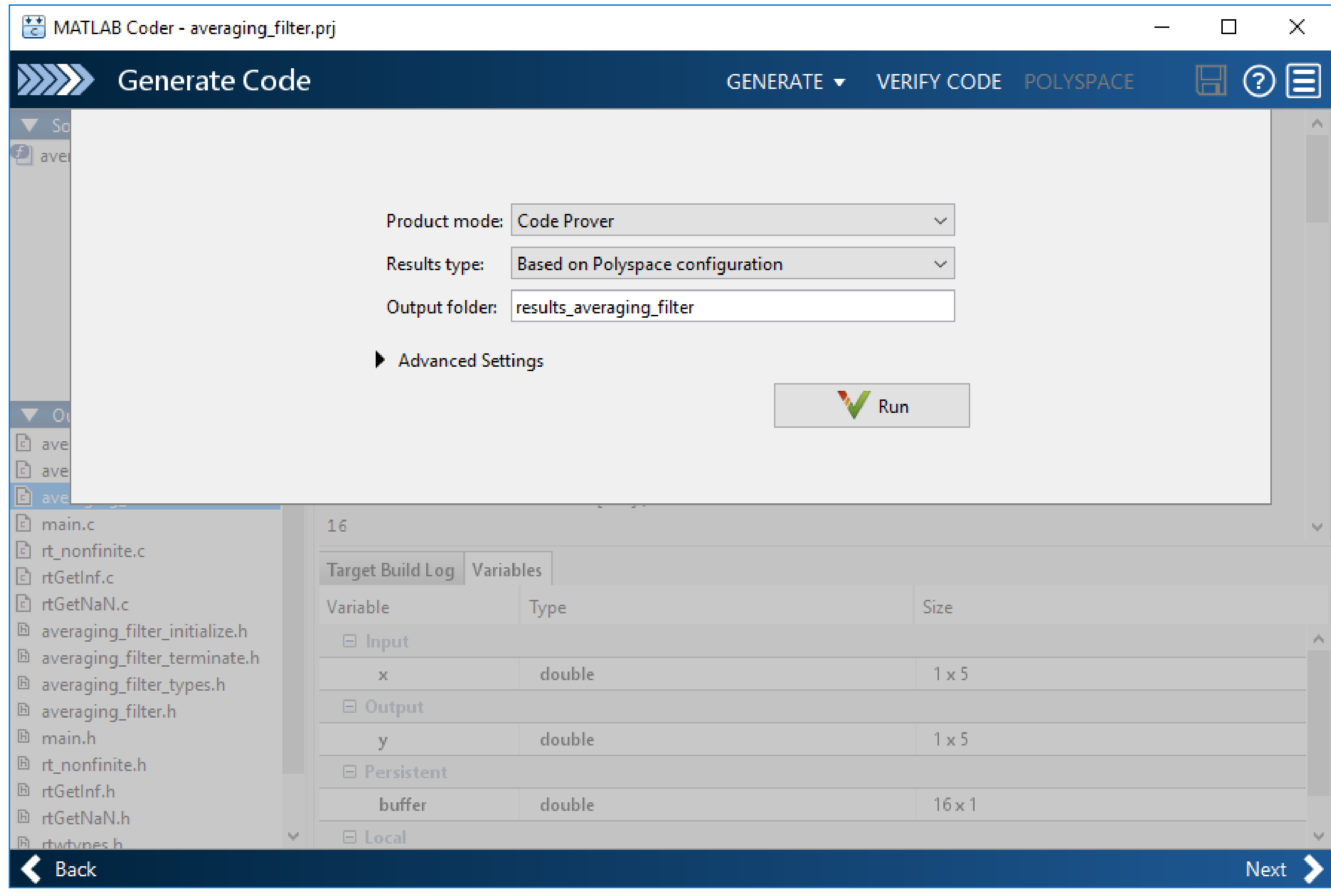
Run Polyspace Analysis

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

1. Generate code.

   From the entry-point function in the file, generate standalone C/C++ code (a static library, dynamically linked library, or executable program) in the MATLAB Coder app. The function has one input. Explicitly specify a data type for the input, for instance, a 1 X 100 vector of type `double`, or provide a file for deriving data types.
2. Analyze the generated code.

   After code generation, open the **Polyspace** pane and click **Run**.



If the analysis is completed without errors, the Polyspace results open automatically. If you close the results, you can reopen them from the final page in the app, under the section **Generated Output**. The results are stored in a subfolder `results_averaging_filter` in the folder containing the MATLAB file.

To script the preceding workflow, run:

```
% Generate code
matlabFileName = fullfile(polyspaceroot, 'help',...
    'toolbox','codeprover','examples','matlab_coder','averaging_filter.m');
codegenFolder = fullfile(pwd, 'codegenFolder');
codegen(matlabFileName, '-config:lib', '-c', '-args', ...
   {zeros(1,100,'double')}, '-d', codegenFolder);

% Configure Polyspace analysis
opts = pslinkoptions('ec');
opts.ResultDir = [tempdir 'results'];
opts.OpenProjectManager = 1;

% Run Polyspace
[polyspaceFolder, resultsFolder] = pslinkrun('-codegenfolder', codegenFolder, opts);
```

Review Analysis Results

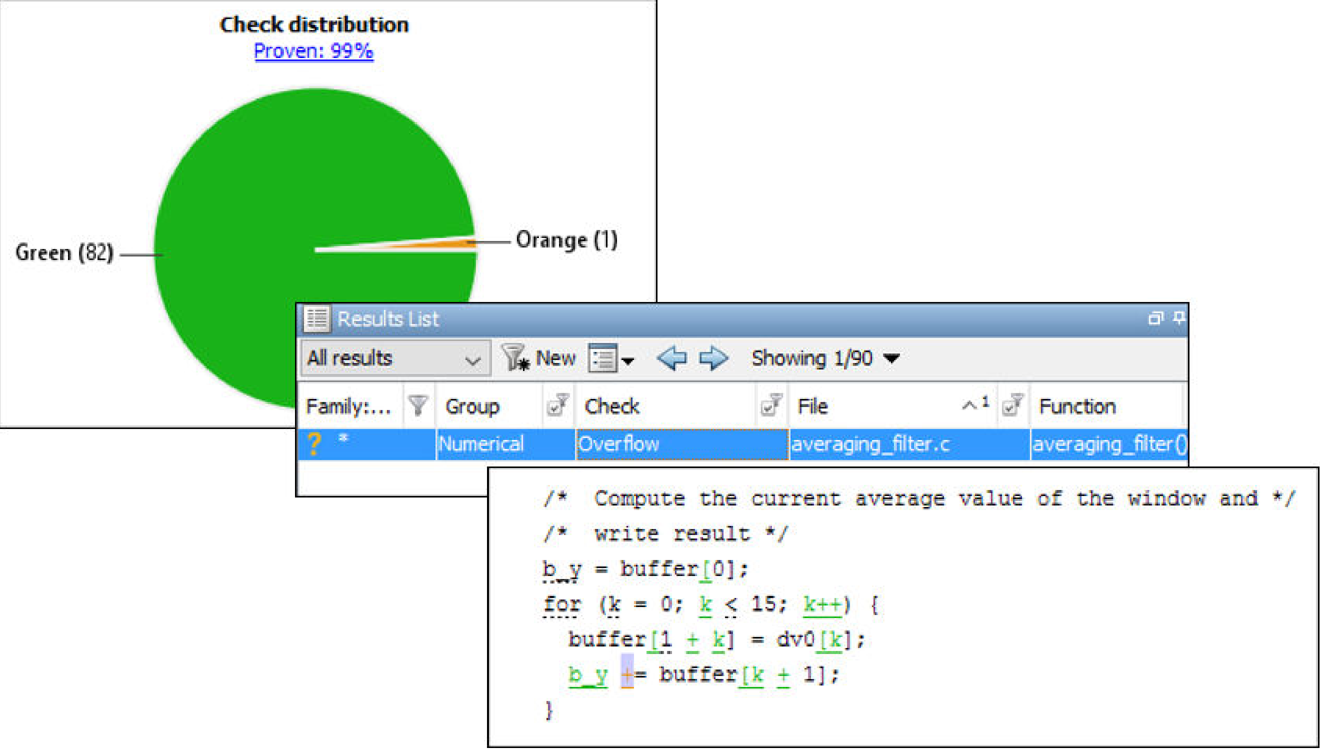
After analysis, the **Results List** pane shows a list of run-time checks. For an explanation of the result colors, see “Code Prover Result and Source Code Colors” on page 32-2.

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                                                   Press 'F2' for focus
  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).

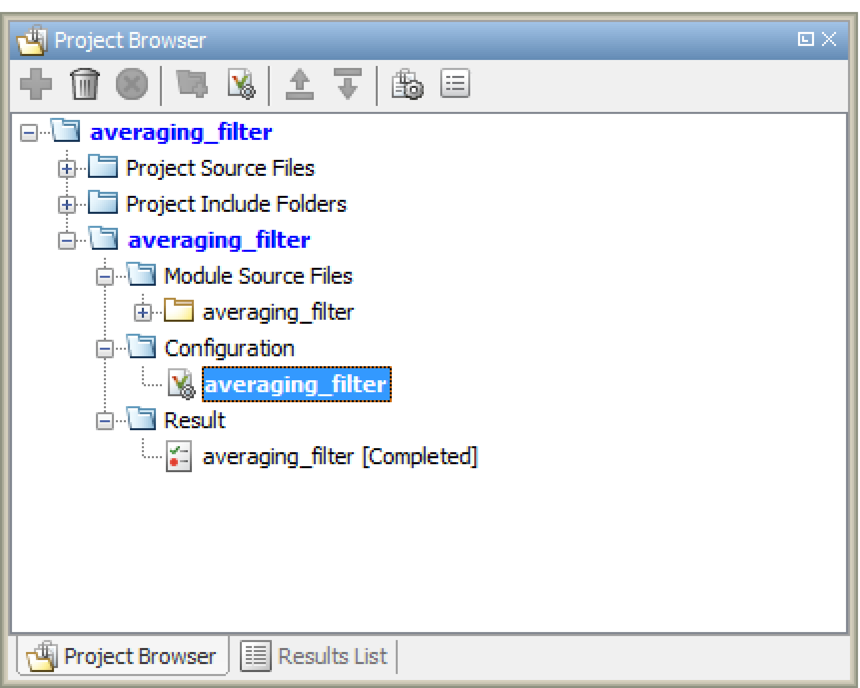
Run Analysis for Specific Design Range

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:

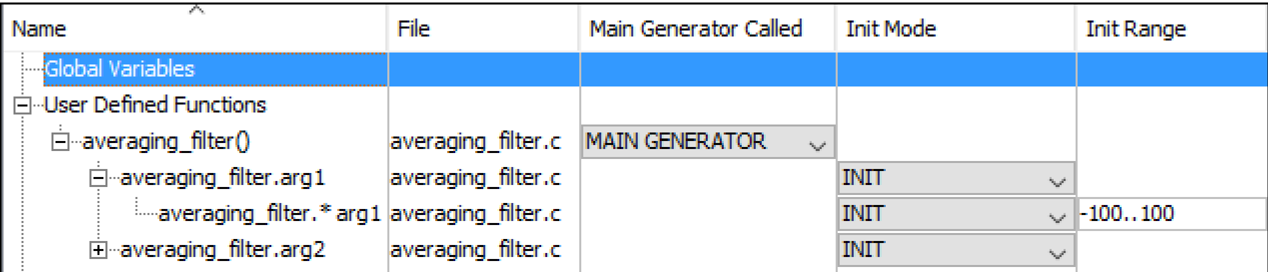
1 Open the analysis configuration.

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.



2 Specify a design range for the inputs.

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

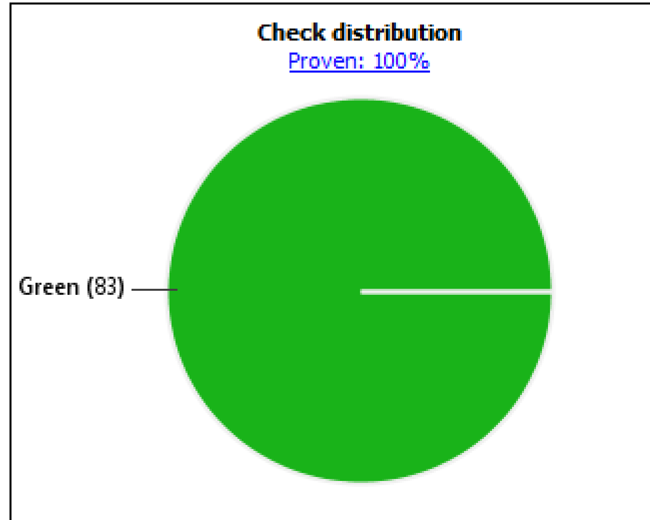


Name	File	Main Generator Called	Init Mode	Init Range
Global Variables				
User Defined Functions				
averaging_filter()	averaging_filter.c	MAIN GENERATOR		
averaging_filter.arg1	averaging_filter.c		INIT	
averaging_filter.* arg1	averaging_filter.c		INIT	-100..100
averaging_filter.arg2	averaging_filter.c		INIT	

You can overwrite the default constraint template or save the constraints elsewhere. For information on the columns in this window, see “External Constraints for Polyspace Analysis” on page 14-6.

3 Rerun the analysis from the Coder app (or at the MATLAB command line) and see the results.

On the **Dashboard** pane, you do not see the previous orange overflow anymore.



See Also

`pslinkrun`

More About

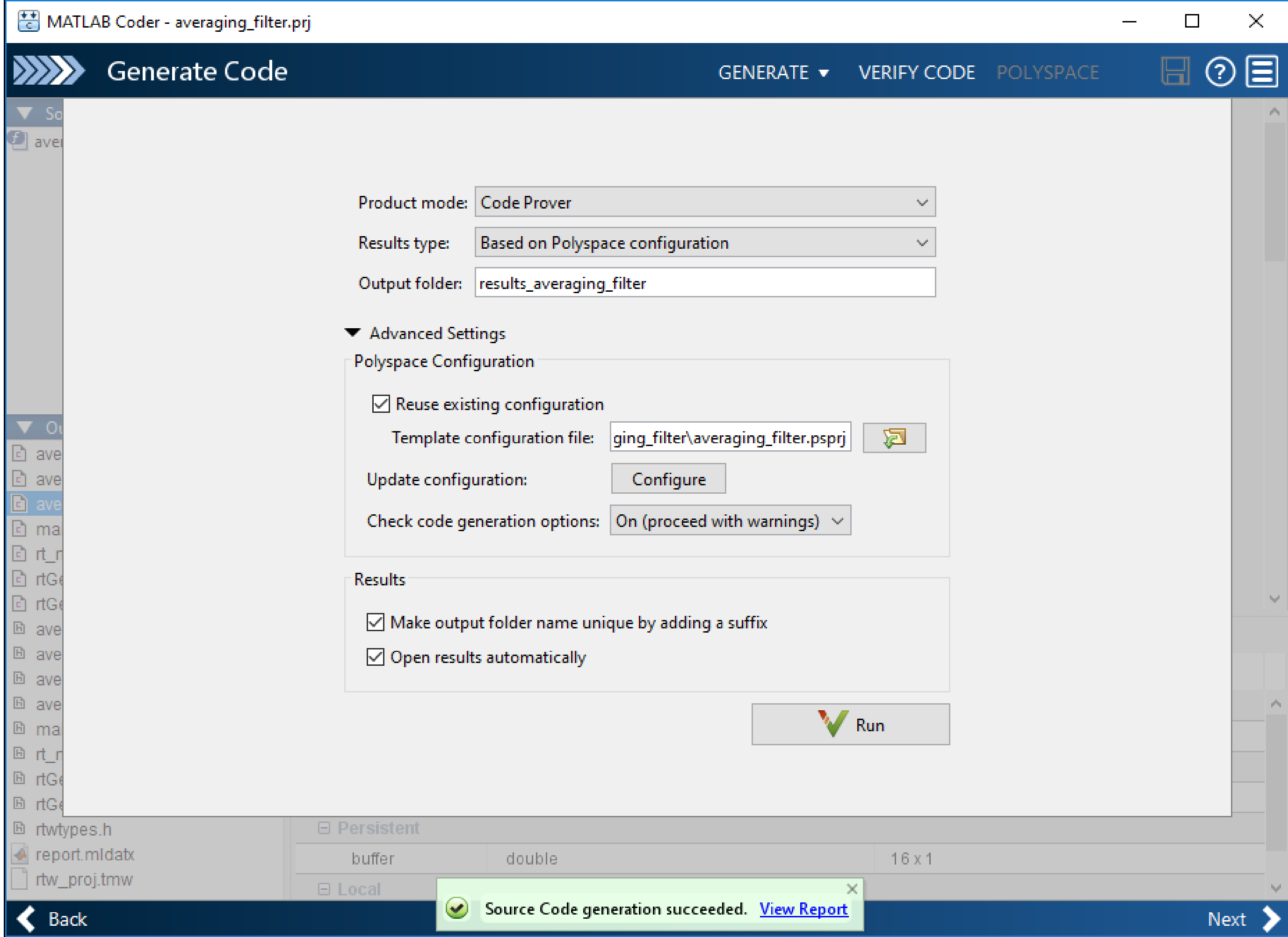
- “Configure Advanced Polyspace Options in MATLAB Coder App” on page 7-7

Configure Advanced Polyspace Options in MATLAB Coder App

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

For getting started with Polyspace analysis in the MATLAB Coder App, see “Run Polyspace on C/C++ Code Generated from MATLAB Code” on page 7-2.

Configure Options



The default analysis runs Code Prover based on a default project configuration. The results are stored in a folder `result_`*`project_name`* in the current working folder.

You can change these options in the MATLAB Coder App itself:

- **Product mode**: Select Code Prover or Bug Finder.
- **Results type**: Check for MISRA C:2004 (MISRA AC AGC) or MISRA C:2012 rule violations, in addition to or instead of the default checkers.

- **Output folder**: Choose an output folder name. To save the results of each run in a new folder, under **Advanced Settings**, select **Make output folder name unique by adding a suffix**.
- **Check code generation options**: Choose to see warnings or errors if the code generation uses options that can result in imprecise Code Prover analysis.

  For instance, if the code generation setting **Use memset to initialize floats and doubles to 0.0** is disabled, Code Prover can show imprecise orange checks because of approximations. See “Orange Checks in Polyspace Code Prover” on page 33-2.

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.

Share and Reuse Configuration

If you change some of the default options in the **Configuration** pane, your updated configuration is saved as a `.psprj` file in the results folder. Using this file, you can reuse your configuration across multiple MATLAB Coder projects.

Reuse Configuration in Coder App

To reuse a previous configuration in the current project opened in the MATLAB Coder App, under **Advanced Settings**, select **Reuse existing configuration**. For **Template configuration file**, provide the `.psprj` file that stores the previous configuration.

The **Results type** option in the MATLAB Coder app still shows **Based on Polyspace configuration** but the configuration used is the one that you provided.

Reuse Configuration on Command Line

At the MATLAB command line, you create an options object with the `pslinkoptions` function. You modify the analysis options by using the properties of this object and then run analysis with the `pslinkrun` function.

```
opts = pslinkoptions('ec');
...
pslinkrun('-codegenfolder', codegenFolder, opts);
```

You can associate advanced analysis options set in a `.psprj` file with the options object. Use the properties `EnablePrjConfigFile` and `PrjConfigFile`.

```
opts.EnablePrjConfigFile = true;
opts.PrjConfigFile = 'C:\Polyspace\config.psprj';
```

For more information, see pslinkoptions Properties.

See Also

`pslinkoptions`

More About

- “Run Polyspace on C/C++ Code Generated from MATLAB Code” on page 7-2

Running Polyspace on AUTOSAR Code

- “Using Polyspace in AUTOSAR Software Development” on page 8-2
- “Choose Between Component-Based and Integration Analysis of AUTOSAR Code with Polyspace” on page 8-5
- “Benefits of Polyspace for AUTOSAR” on page 8-7
- “Run Polyspace on AUTOSAR Code” on page 8-14
- “Select AUTOSAR XML (ARXML) and Code Files for Polyspace Analysis” on page 8-19
- “Troubleshoot Polyspace Analysis of AUTOSAR Code” on page 8-22
- “Interpret Errors and Warnings in Polyspace Analysis of AUTOSAR Code” on page 8-26
- “Run Polyspace on AUTOSAR Code with Conservative Assumptions” on page 8-29
- “Run Polyspace on AUTOSAR Code Using Build Command” on page 8-31

Using Polyspace in AUTOSAR Software Development

Whatever your role in the AUTOSAR software development workflow, you can benefit from using a static code analysis tool such as Polyspace.

Polyspace supports two approaches for verifying AUTOSAR code:

- Component-based analysis

  In this approach, you provide your AUTOSAR design specifications in ARXML format. The analysis reads these specifications, creates a separate C code module for each software component, and then checks each module for run-time errors and mismatch with design specifications.

  For an overview of this approach, see "Benefits of Polyspace for AUTOSAR" on page 8-7.
- Integration analysis

  In this approach, you do not provide the design specifications but simply run Code Prover on a single project with all relevant source code. To make the analysis AUTOSAR-aware, use the value `autosar` for the analysis option `Libraries used (-library)`.

See also "Choose Between Component-Based and Integration Analysis of AUTOSAR Code with Polyspace" on page 8-5.

Check if Implementation of Software Components Follow Specifications

This check is supported only in the component-based analysis.

Suppose you are part of an OEM specifying the structure and runtime behavior of the software components in the application layer, including the data types, events and runnables. You want to check if the tier-1 suppliers providing the code implementation of the software components follow your specifications.

Check the code implementation of each software component individually or see an overview of results for all software component implementations. To see an overview:

1. Run Polyspace on all software components and upload all results to Polyspace Access.
2. In the results, see if:

   - All runnables are implemented. See if the checker **AUTOSAR runnable not implemented** shows any result.
   - All runnables implementations conform to data constraints in the specifications. See if the checker **Invalid result of AUTOSAR runnable implementation** shows any result.
   - Arguments to `Rte_` functions follow data constraints in the specifications. See if the checker **Invalid use of AUTOSAR runtime environment function** shows any result.
   - There are other possibilities of run-time errors.

To begin checking the code implementation of software components against ARXML specifications:

1. Provide the locations of your ARXML and code folders. Run Polyspace to check the code implementation of all software components against ARXML specifications.

If you run verification on a remote server, you can specify that all results must be uploaded to Polyspace Access after verification. Otherwise, you can upload them later.

See “Run Polyspace Code Prover on Server and Upload Results to Web Interface”.

2. Upload all results to Polyspace Access. When uploading, make sure you use the same project name and version number for all results.

   See “Upload Results to Polyspace Access” on page 2-28.

3. In Polyspace Access, click the project name in the **Project Explorer** and see a summary of the results.

   See “Dashboard in Polyspace Access Web Interface” on page 26-7.

Alternatively, you can ask for code analysis reports from the suppliers. The reports are produced individually for each software component. To begin, see “Generate Reports from Polyspace Results” on page 25-2.

Assess Impact of Edits to Specifications

This check is supported only in the component-based analysis.

Suppose you are part of an OEM and want to add to or edit the specifications that you provide to a tier-1 supplier. Before making the edits, you want to test their potential impact on the existing code implementation.

Check the code implementation of software components that are likely to be impacted. Compare Code Prover analysis results that use the modified specifications with results that use the original specifications.

To begin comparing verification results for a software component:

1. Run Polyspace using the original specifications.

   See “Run Polyspace on AUTOSAR Code” on page 8-14.

2. Upload the result for a software component to Polyspace Access.

   See “Upload Results to Polyspace Access” on page 2-28.

3. Rerun Polyspace using the updated specifications.
4. Upload the new result to Polyspace Access.
5. See if there is an increase in the number of red, gray or orange checks.

   See “Compare Results in Polyspace Access Project to Previous Runs and View Trends” on page 28-23.

Check Code Implementation for Run-time Errors and Mismatch with Specifications

The check for run-time errors is supported both in the component-based analysis and integration analysis. The check for mismatch with design specifications is supported only in the component-based analysis.

Suppose you are part of a tier-1 supplier providing the code implementation of software components based on specifications from an OEM. You want to check for run-time errors such as overflow and division by zero or violations of data constraints in the ARXML specifications.

Check software components that you implemented. Use the advanced option `-autosar-behavior` to check specific software components.

To begin:

1. Run Polyspace on the code implementation of your software components.
2. If you update the implementation of a software component, you can continue to use the same project to reanalyze your code. The later analysis only consider the software components whose implementation changed since the previous analysis.

See “Run Polyspace on AUTOSAR Code” on page 8-14.

Check Code Implementation Against Specification Updates

This check is supported only in the component-based analysis.

Suppose you are part of a tier-1 supplier implementing specifications from an OEM. You receive some updates to the specifications. If you had been running Polyspace to compare your code against the specifications, you can quickly check if the specification changes introduced any errors.

In this case, you will already have set up your project, possibly with additional options to emulate your compiler. You can reuse these options when creating a new project from the new ARXML specifications.

See Also

More About

- “Benefits of Polyspace for AUTOSAR” on page 8-7
- “Run Polyspace on AUTOSAR Code” on page 8-14
- “Review Polyspace Results on AUTOSAR Code” on page 22-80

Choose Between Component-Based and Integration Analysis of AUTOSAR Code with Polyspace

Polyspace supports two approaches for verifying AUTOSAR code: component-based analysis and integration analysis.

Definitions

In a nutshell, a component-based analysis requires the design specifications of AUTOSAR software components and extracts all required information from these specifications, while an integration analysis does not require these specifications.

- In the *component-based analysis* approach, you provide your AUTOSAR design specifications in ARXML format. The analysis reads these specifications, creates a separate C code module for each software component, and then checks each module for run-time errors and mismatch with design specifications.

  For an overview of this approach, see “Benefits of Polyspace for AUTOSAR” on page 8-7. For information on how to set up a component-based analysis, see “Run Polyspace on AUTOSAR Code” on page 8-14.
- In the *integration analysis* approach, you do not provide the design specifications but simply run Bug Finder or Code Prover on a single project with all relevant source code. To make the analysis AUTOSAR-aware, use the value `autosar` for the analysis option `Libraries used (-library)`.

Similarities and Differences

Both the component-based and the integration analysis adhere to the AUTOSAR standard. In addition, the component-based analysis also uses the design specifications of your AUTOSAR project.

Similarities

In both forms of analysis, functions from the AUTOSAR Runtime Environment or RTE layer are replaced with precise stubs. These stubs allow:

- *Faster and more precise analysis*, since the analysis does not attempt to check the implementation of the RTE functions. Instead, the analysis emulates the RTE functions based on the AUTOSAR standard. (Checking the function implementations is not necessary for verifying function usage and might result in precision loss.)
- *Run-time checks on function arguments* that look for violations of the AUTOSAR standard.

  For instance, according to the standard definition of an input argument, if the argument is a pointer, the pointer must point to an initialized buffer. Polyspace can check whether the argument in a given call points to a possibly noninitialized buffer.

  Both the checks `Non-compliance with AUTOSAR specification` (integration analysis) and `Invalid use of AUTOSAR runtime environment function` (component-based analysis) look for violations of the AUTOSAR standard.

Differences

In addition to using precise stubs for the RTE layer, the component-based analysis also uses information from the design specifications that you provide. As a result, the component-based analysis has these additional features not found in the integration analysis:

- Automatic splitting of code into modules based on software component specifications.
- Automatic identification of AUTOSAR runnables in each module (entry points) and run-time checks on these runnables. See also `Invalid result of AUTOSAR runnable implementation`.
- Additional checks on RTE function arguments to determine if they violate data constraints from the design specifications. These checks can be more precise than the run-time checks based on the AUTOSAR standard since they use the narrower data constraints on specific data types from the design specifications. See also `Invalid use of AUTOSAR runtime environment function`.

Choosing Between Component-Based and Integration Analysis

If you simply want to perform an analysis that is aware of specifications from the AUTOSAR standard but do not want to compare the code with your design specifications, you can perform the integration analysis. Otherwise, perform the component-based analysis to keep your software component implementations in sync with your design specifications.

The component-based analysis can provide richer results than the integration analysis, but requires a more elaborate setup:

- The component-based analysis runs in three phases: ARXML parsing, code extraction, and finally static code verification. Errors in each phase can propagate to further downstream issues. To work around the errors, you might require detailed knowledge of your AUTOSAR project. For instance, you might have to exclude artifacts such as leftover template files from code generators.

  Only Code Prover supports a component-based analysis.
- The integration analysis works only with the code provided and is simpler to set up. You have to run only a regular analysis with one additional option enabled to make the analysis AUTOSAR-aware.

  Both Bug Finder and Code Prover support an integration analysis with the same checks. Bug Finder is less exhaustive than Code Prover and might show fewer results for these checks.

See Also

More About

- “Using Polyspace in AUTOSAR Software Development” on page 8-2
- “Benefits of Polyspace for AUTOSAR” on page 8-7

Benefits of Polyspace for AUTOSAR

This topic describes a component-based approach to verifying AUTOSAR code with Polyspace. For an integration analysis approach, see “Choose Between Component-Based and Integration Analysis of AUTOSAR Code with Polyspace” on page 8-5.

Polyspace for AUTOSAR runs static program analysis on code implementation of AUTOSAR software components. The analysis looks for possible run-time errors or mismatch with specifications in the AUTOSAR XML (ARXML).

Polyspace for AUTOSAR reads the ARXML specifications that you provide and modularizes the analysis based on the software components in the ARXML specifications. The analysis then checks each module for:

- Mismatch with AUTOSAR specifications: These checks aim to prove that certain functions are implemented or used in accordance with the specifications in the ARXML. The checks apply to runnables (functions provided by the software components) and to the usage of functions supplied by the Run-Time Environment (RTE). See also:

  - `AUTOSAR runnable not implemented`
  - `Invalid result of AUTOSAR runnable implementation`
  - `Invalid use of AUTOSAR runtime environment function`

  For instance, if an RTE function argument has a value outside the constrained range defined in the ARXML, the analysis flags a possible issue.
- Run-time errors: These checks aim to prove the absence of certain types of run-time errors in the bodies of the runnables (for instance, overflow). The proof uses the specifications in the ARXML to determine precise ranges for runnable arguments and RTE function return values and output arguments. For instance, the proof considers only those values of runnable arguments that are specified in their AUTOSAR data types.

After analysis, you can open the results for each module in the Polyspace user interface. When reviewing a mismatch between code and ARXML specifications, you can navigate to the relevant extract of the ARXML.

This topic shows how Polyspace is AUTOSAR-aware and helps in the AUTOSAR development workflow. For the actual steps for running Polyspace, see:

- “Run Polyspace on AUTOSAR Code” on page 8-14
- “Review Polyspace Results on AUTOSAR Code” on page 22-80

Polyspace Modularizes Analysis Based on AUTOSAR Components

Polyspace for AUTOSAR modularizes your code by reusing the modularization already present in your ARXML specifications. The modularization is based on the software components in the ARXML specifications. Modularizing your code is essential to avoid long analysis times and allow more precise analysis.

A software component consists of one or more runnables. You implement runnables through functions.

A software component (SWC) is the unit of functionality in the application layer of the AUTOSAR architecture. A software component has an internal behavior that consists of data types, events, one or more runnable entities (tasks), and other information.

The AUTOSAR XML lists the internal behavior of a software component like this (AUTOSAR XML schema version 4.0):

```
<APPLICATION-SW-COMPONENT-TYPE>
  <SHORT-NAME>swc001</SHORT-NAME>
  <INTERNAL-BEHAVIORS>
    <SWC-INTERNAL-BEHAVIOR>
      <SHORT-NAME>bhv001</SHORT-NAME>
      <DATA-TYPE-MAPPING-REFS>
       ...
      </DATA-TYPE-MAPPING-REFS>
      <EVENTS>
       ...
      </EVENTS>
      <RUNNABLE-ENTITY>
         <SHORT-NAME>foo</SHORT-NAME>
         ...
       </RUNNABLE-ENTITY>
    </SWC-INTERNAL-BEHAVIOR>
  </INTERNAL-BEHAVIORS>
<APPLICATION-SW-COMPONENT-TYPE>
```

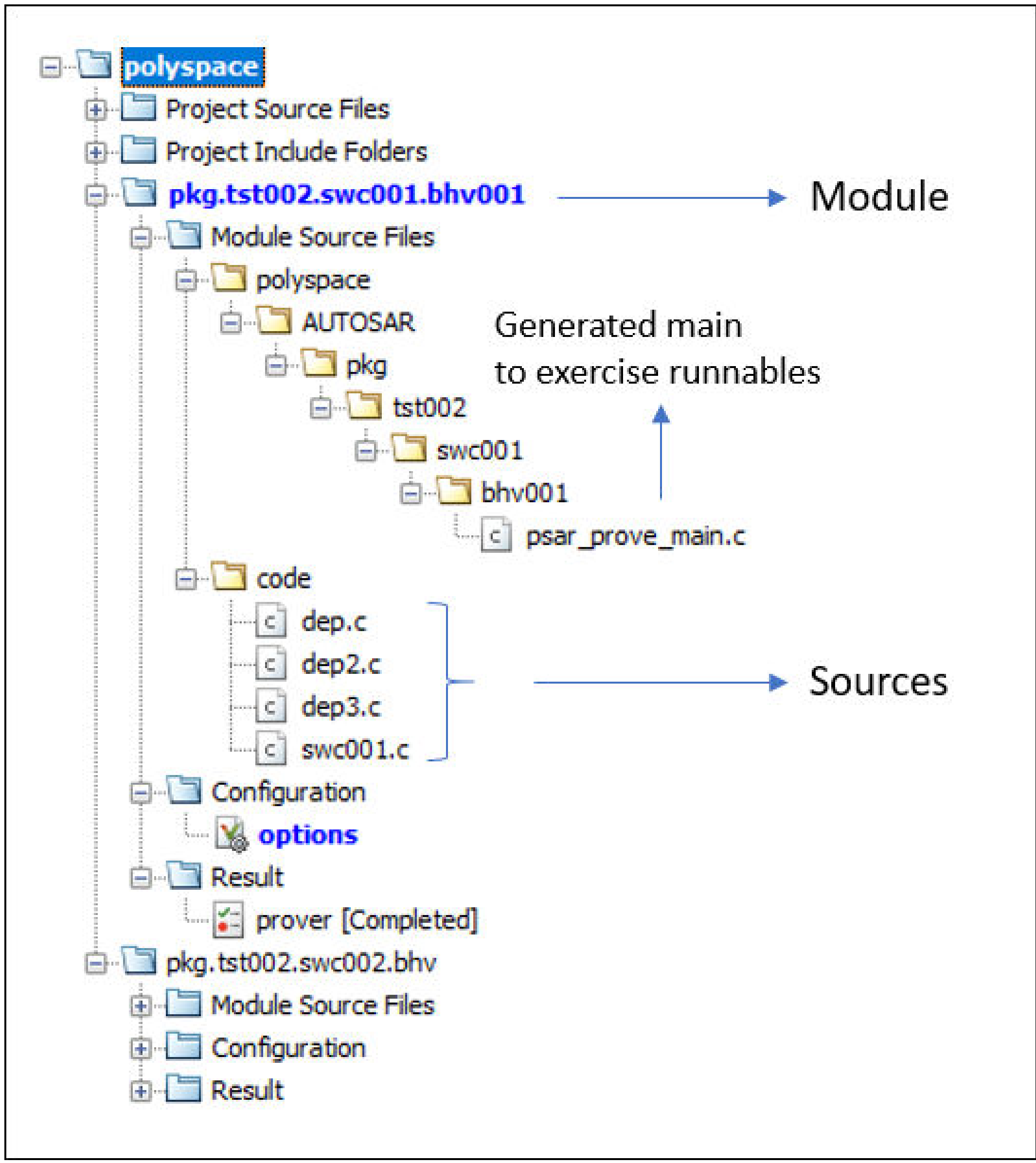
As a developer, you implement the bodies of these runnable entities through handwritten C functions or functions generated from a Simulink model.

```
iOperations_ApplicationError foo(
    Rte_Instance const self,
    app_Array_2_n320to320ConstRef aInput,
    app_Array_2_n320to320Ref aOutput,
    app_Enum001Ref aOut2)
{
  /* Your implementation */
}
```

Polyspace collects the source code for each software component into a module.

Using the information in the AUTOSAR XML, Polyspace for AUTOSAR creates a project with a separate module for each software component. In a single module, Polyspace collects the source code (`.c` and `.h` files) containing the implementation of all runnables in the software component and generates any additional header file required for the implementation.

A Polyspace project with two modules from two software components can look like this:



The module name corresponds to the fully qualified name of the internal behavior of the software component.

For instance, the name `pkg.tst002.swc001.bhv001` corresponds to this XML structure (AUTOSAR XML schema version 4.0):

```
<AR-PACKAGE>
  <SHORT-NAME>pkg</SHORT-NAME>
  <AR-PACKAGES>
    <AR-PACKAGE>
      <SHORT-NAME>tst002</SHORT-NAME>
      <ELEMENTS>
        <APPLICATION-SW-COMPONENT-TYPE>
          <SHORT-NAME>swc001</SHORT-NAME>
            ...
```

```
          <SWC-INTERNAL-BEHAVIOR>
             <SHORT-NAME>bhv001</SHORT-NAME>
              ...
          </SWC-INTERNAL-BEHAVIOR>
        </APPLICATION-SW-COMPONENT-TYPE>
       </ELEMENTS>
    </AR-PACKAGE>
  </AR-PACKAGES>
</AR-PACKAGE>
```

If `bhv001` has one runnable `foo`, Polyspace collects the files containing the function `foo` and the functions called in `foo` into one module.

For this modularization, you simply provide the two folders with ARXML and source files.

Polyspace for AUTOSAR uses the fact that the required information is already present in your ARXML specifications and modularizes your code. You do not need to know the details of the ARXML specifications or code implementation for running the analysis. You simply provide the folders containing your ARXML and source files.

Without this automatic modularization, you have to manually add the implementation of each software component (the files with the entry point functions implementing runnables, the functions called within, and so on) to a module. Not only that, you have to define the interface for each runnable, that is, the range of values for inputs based on their data types.

Polyspace Detects Mismatch Between Code and AUTOSAR XML Spec

Polyspace for AUTOSAR detects mismatch between the ARXML specifications of AUTOSAR software components and their code implementation. The mismatch can occur at run time between data constraints in the ARXML and actual values of function arguments in the code. The mismatch detection occurs for certain functions only: functions implementing the runnables and `Rte_` functions used in the runnables. The arguments of these functions have data types specified in the ARXML.

AUTOSAR runnables communicate via `Rte_` functions.

The implementation of an AUTOSAR runnable uses functions provided by the run-time environment (RTE) for communication with runnables in other SWCs. For instance, the function `Rte_IWrite_`*`runnable_port_variable`* can be used to provides write access to *`variable`* from the current runnable.

```
Rte_IWrite_step_out_e4(self, e4);
```

The function arguments have data types specified in the ARXML.

These functions have signatures specified in the AUTOSAR standard with parameter data types that are detailed in ARXML specifications. For instance, the standard defines the signature of the `Rte_IWrite_` function like this, where the type of *`data`* is specified in the ARXML.

```
void Rte_IWrite_re_p_o([IN Rte_Instance], IN data)
```

When deploying your implementation, an Run-Time Environment generator uses the information in the ARXML specifications to create header files with data type definitions for your application. When developing your implementation, you do not have to worry about details of communication with other SWCs. You simply use the `Rte_` functions and the data types provided for your implementation.

Likewise, the data types of the inputs, outputs and return value of your runnable are also listed in the ARXML.

You can constrain data types in the ARXML using data constraints.

In your ARXML specifications, you often limit the values associated with data types using data constraints. A data constraint specification can look like this (AUTOSAR XML schema version 4.0):

```
<APPLICATION-PRIMITIVE-DATA-TYPE>
  <SHORT-NAME>Float_n100p4321to100p8765</SHORT-NAME>
  <CATEGORY>VALUE</CATEGORY>
  <SW-DATA-DEF-PROPS>
  ...
    <DATA-CONSTR-REF DEST="DATA-CONSTR">n320to320</DATA-CONSTR-REF>
..</SW-DATA-DEF-PROPS>
</APPLICATION-PRIMITIVE-DATA-TYPE>
...
<DATA-CONSTR>
  <SHORT-NAME>n320to320</SHORT-NAME>
  <DATA-CONSTR-RULES>
     <DATA-CONSTR-RULE>
        <PHYS-CONSTRS>
           <LOWER-LIMIT INTERVAL-TYPE="CLOSED">-320</LOWER-LIMIT>
           <UPPER-LIMIT INTERVAL-TYPE="CLOSED">320</UPPER-LIMIT>
           <UNIT-REF DEST="UNIT">/pkg/types/units/NoUnit</UNIT-REF>
        </PHYS-CONSTRS>
     </DATA-CONSTR-RULE>
  </DATA-CONSTR-RULES>
</DATA-CONSTR>
```

When an `Rte_` function uses data types that are constrained this way, the expectation is that values passed to the function stay within the constrained range. For instance, for the preceding constraint, if an `Rte_IWrite_` function uses a variable of type `n320to320`, its value must be within [-320, 320].

If you generate the ARXML in Simulink, the data constraints come from signal ranges in the model.

At run time, your code implementation can violate data constraints.

The `Rte_` functions represent ports in the SWC interface. So, in effect, when you constrain the data type of an argument in the ARXML, the ports are prepared for data within that range. However, in your code implementation, when you invoke an `Rte_` function, you can pass an argument outside a constrained range.

For instance, in this call to `Rte_IWrite_step_out_e4`:

```
Rte_IWrite_step_out_e4(self, e4);
```

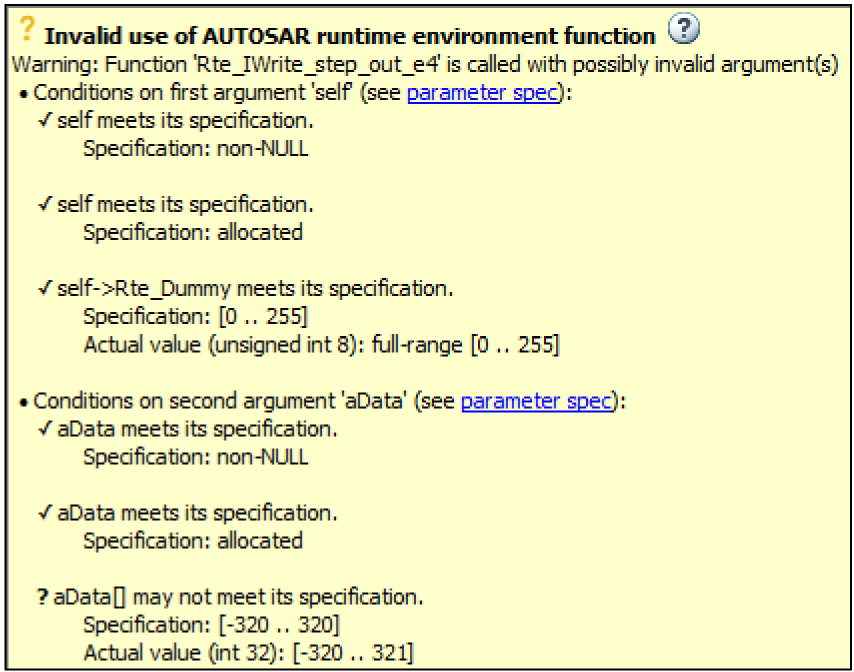
the second argument of `Rte_IWrite_step_out_e4` can have the previously defined data type `n320to320`. But at run time, your code implementation can pass a value outside the range [-320, 320]. The argument might be the result of a series of previous operations and one of those operations can cause the out-of-range value.

```
app_Enum001 e4;
e4 = Rte_IRead_step_in_e4(self);
...
/* Some operation on e4*/
...
Rte_IWrite_step_out_e4(self, e4);
```

Polyspace Code Prover checks for possible data constraint violations.

You can either test each invocation of an `Rte_` function to check if the arguments are within the constrained range and also make sure that the tests cover all execution paths in the runnable. Alternatively, you can use static analysis that guarantees that all execution paths leading up to the `Rte_` function call are considered (up to certain reasonable assumptions on page 32-10). Polyspace uses static analysis to determine if arguments to `Rte_` functions stay within the constrained range defined in the ARXML files.

The checks for mismatch detection in a Polyspace analysis can show results like this. Here, the second argument in the invocation of `RTE_IWrite_step_out_e4` violates the data constraints in the ARXML specifications.



See Also

`Invalid result of AUTOSAR runnable implementation` | `Invalid use of AUTOSAR runtime environment function`

More About

- "Using Polyspace in AUTOSAR Software Development" on page 8-2
- "Run Polyspace on AUTOSAR Code" on page 8-14
- "Review Polyspace Results on AUTOSAR Code" on page 22-80

Run Polyspace on AUTOSAR Code

This topic describes a component-based approach to verifying AUTOSAR code with Polyspace. For an integration analysis approach, see “Choose Between Component-Based and Integration Analysis of AUTOSAR Code with Polyspace” on page 8-5.

Polyspace for AUTOSAR runs static program analysis on code implementation of AUTOSAR software components. The analysis looks for possible run-time errors or mismatch with specifications in the AUTOSAR XML (ARXML).

To run Polyspace on code implementation of AUTOSAR software components, provide this information:

- ARXML folder: This folder contains all the `.arxml` files that define your AUTOSAR model. The files specify the data types, runnables, events and other information about the software components in your AUTOSAR model.

  Note that Polyspace can parse an AUTOSAR XML schema only for releases 4.0 and later.
- Source code folder: This folder contains the C code implementation of the software components. The `.c` files in this folder contain functions implementing the AUTOSAR runnables and other called functions. The folder can also contain header files referenced in your source files.

  If you reference header files located in another folder, you can provide that location separately.

The analysis parses your ARXML files, reads your source files and creates a Polyspace project with a separate module for each software component. Polyspace Code Prover then checks each module for run-time errors or violations of data constraints in the ARXML at run-time.

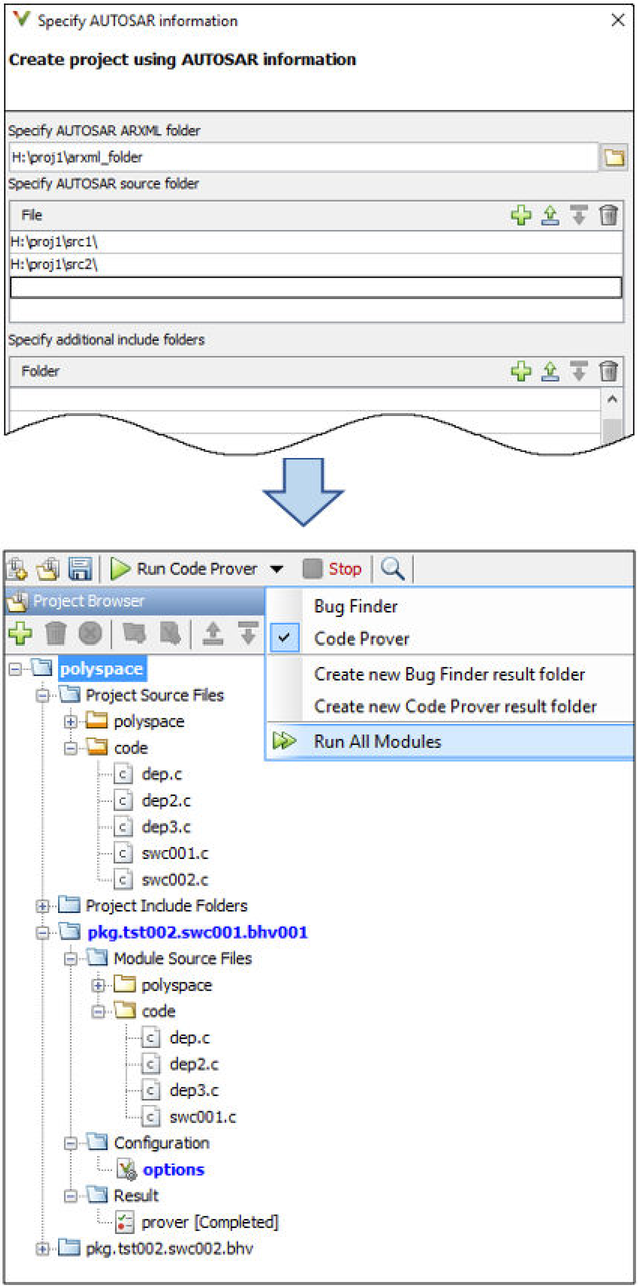
This topic shows how to run Polyspace on code implementation of AUTOSAR software components. You can run Polyspace from the user interface of the Polyspace desktop products or the command line:

- In the user interface, the analysis happens in two steps: creating a Polyspace project from the ARXML and code folders and running Code Prover on the project.
- At the command line, the analysis can be done in one shot with the `polyspace-autosar` command.

To follow the steps in this tutorial, use the demo files in *`polyspaceroot`*`\help\toolbox\codeprover\examples\polyspace_autosar`.

Run Polyspace in User Interface

In the Polyspace desktop products, you can create a Polyspace project in the user interface. Each module in the project contains the source files implementing one software component. You can run verification on a single module or all modules together.



Read ARXML and Sources

Specify upfront that the project must be created from AUTOSAR specifications.

1. Select **File > New**. In the Project-Properties window, select **Create from AUTOSAR specification**.
2. In the field **Specify AUTOSAR ARXML folder**, specify the top level folder containing your ARXML files. In the section **Specify AUTOSAR source folder**, specify all the folders containing C/C++ source files.
3. Click **Run**.

The software parses your ARXML specifications and C code implementation and creates a Polyspace project. Each module in the project references C files that implement one software component. The module name corresponds to the fully qualified name of the software component, as specified in the ARXML. See “Benefits of Polyspace for AUTOSAR” on page 8-7.

If the software fails to parse your ARXML specifications or runs into compilation issues with your code, see additional details in the **Command output** or **Project status** tab. Investigate the issue further and fix your ARXML files or code accordingly. See “Troubleshoot Polyspace Analysis of AUTOSAR Code” on page 8-22.

In some cases, you might have to provide additional paths to include folders or macro definitions to troubleshoot errors.

- To specify paths to include files that are not directly under the source folder, use the field **Specify additional include folders**.

  This field corresponds to the option `-I` of `polyspace-autosar`.
- To specify data type and macro definitions that are not in your source files, use the field **Specify additional macro definitions**. Specify a file with the definitions.

  This field corresponds to the option `-include` of `polyspace-autosar`.
- To specify one of the advanced command-line options associated with `polyspace-autosar`, use the field **Advanced settings**.

  For instance:

  - You might want to specify a compiler and target architecture. By default, compilation of projects created from AUTOSAR specifications use the gnu4.7 compiler and i386 architecture.

    To specify a visual11.0 compiler with x86_64 architecture, enter this option:

    ```
    -extra-project-options "-compiler visual11.0 -target x86_64"
    ```

    See also `Compiler (-compiler)` and `Target processor type (-target)`.

Configure Project

Once a project is created, you can change some of the default analysis options. For instance, you can generate a report after analysis using the options in the **Reporting** section. For details on how to specify options, see “Specify Polyspace Analysis Options” on page 12-2.

You do not need the options in these sections for a project generated from an AUTOSAR description:

- “Inputs and Stubbing”: External data constraints in your ARXML files are extracted automatically when you create a Polyspace project. You cannot explicitly specify external constraints.
- “Multitasking”: You cannot perform a multitasking analysis with the Polyspace project because each module analyzes the implementation of one software component. To detect data races, create a separate project for the entire application and explicitly add your source folders. Specify the ARXML files relevant for multitasking and run Bug Finder. For more information, see `ARXML files selection (-autosar-multitasking)`.
- “Code Prover Verification”: A `main` function is generated (in the file `psar_prove_main.c`) when you create a Polyspace project from an AUTOSAR description. The `main` function calls functions that implement runnable entities in the software components. The generated `main` is needed for the Code Prover analysis. You cannot change the properties of this `main` function.

Verify Code

Verify each module individually or all the modules. The verification of a module checks the code implementation of the corresponding software component against the ARXML specifications and also checks for run-time errors. See “Benefits of Polyspace for AUTOSAR” on page 8-7.

To verify a single module, select the module and click **Run Code Prover**. To verify all modules, from the drop down list beside **Run Code Prover**, select **Run All Modules**.

Update Project for Later Changes

If you update your code or ARXML specifications, you can reanalyze the modules. To begin, right-click your project and select **Update AUTOSAR Project**. Recreate your project and rerun verification on the modules.

If you change the code only for specific software components, only the affected modules are recreated. The modules corresponding to the other software components remain unchanged.

Additional Information

See “Review Polyspace Results on AUTOSAR Code” on page 22-80.

Run Polyspace Using Scripts

Run the `polyspace-autosar` command with paths to your ARXML and source code folder. The command parses the ARXML and source files, creates a Polyspace project and analyzes all modules in the project for run-time errors or violation of data constraints in the ARXML.

In the first run, specify the path to your ARXML and source files explicitly. In later runs, specify the file `psar_project.xhtml` created in the previous run. The analysis detects changes in the ARXML and source files since the last run and reanalyzes only those modules where the software component implementation changed. If the ARXML specification changed since the previous analysis, the new analysis reanalyzes all modules.

For instance, you can run these commands in a `.bat` script. In the first run, this script looks for the ARXML specifications in a folder `arxml` in the current folder, and C source files in a folder `code`. The results are stored in a folder `polyspace` in the current folder. In later runs, the analysis reuses the result from the previous run through the file `psar_project.xhtml` and updates the results only for the software components modified since the last run.

```
echo off
set POLYSPACE_AUTOSAR_PATH=C:\Program Files\Polyspace\R2019a\polyspace\bin

IF NOT EXIST polyspace\psar_project.xhtml (
"%POLYSPACE_AUTOSAR_PATH%\polyspace-autosar" -create-project polyspace \
                   -arxml-dir arxml -sources-dir code
) ELSE (
"%POLYSPACE_AUTOSAR_PATH%\polyspace-autosar" \
                   -update-project polyspace\psar_project.xhtml
)
Pause
```

You can also run Code Prover on code implementation of AUTOSAR software components with MATLAB scripts. See `polyspaceAutosar`.

Additional Information

See "Review Polyspace Results on AUTOSAR Code" on page 22-80.

See Also

`AUTOSAR runnable not implemented` | `Invalid result of AUTOSAR runnable implementation` | `Invalid use of AUTOSAR runtime environment function`

More About

- "Benefits of Polyspace for AUTOSAR" on page 8-7
- "Using Polyspace in AUTOSAR Software Development" on page 8-2
- "Run Polyspace on AUTOSAR Code with Conservative Assumptions" on page 8-29
- "Select AUTOSAR XML (ARXML) and Code Files for Polyspace Analysis" on page 8-19
- "Review Polyspace Results on AUTOSAR Code" on page 22-80
- "Troubleshoot Polyspace Analysis of AUTOSAR Code" on page 8-22

Select AUTOSAR XML (ARXML) and Code Files for Polyspace Analysis

This topic describes a component-based approach to verifying AUTOSAR code with Polyspace. For an integration analysis approach, see “Choose Between Component-Based and Integration Analysis of AUTOSAR Code with Polyspace” on page 8-5.

If you run Polyspace on an AUTOSAR project by providing only the ARXML and source root folder, you might run into setup errors.

- A typical AUTOSAR root folder contains many extraneous files, for instance, files for testing and documentation. These files might have extensions such as `.arxml` or `.c`, and Polyspace might incorrectly treat them as ARXML or source files and include them in the analysis.
- The implementation of some software components might be in progress and incomplete.

If you are familiar with the structure of your AUTOSAR project, you can exclude extraneous and incomplete files and folders from the analysis.

The `polyspace-autosar` options that support file selections are `-select-arxml-files` and `-select-source-files`. Since the fully qualified names of AUTOSAR behaviors and data types use similar separators as file paths, you can also specify behaviors with `-autosar-behavior` and data types with `-autosar-datatype` using the file selection syntax.

Adapt Linux find Command to Select Files

The file selection syntax closely emulates the Linux `find` command. You specify a root folder followed by file inclusions and exclusions by using shell patterns or regular expressions.

For instance, to find all files with extension `.arxml` in a folder `package`, you use the Linux command:

```
find package -name '*.arxml'
```

To specify all ARXML files in the folder `package`, copy the part of the command following `find`. Provide the copied content as a string argument to the option `-select-arxml-files` of the `polyspace-autosar` command:

```
polyspace-autosar -select-arxml-files "package -name '*.arxml'"
```

In other words, you select ARXML files by specifying the root folder plus `find` options as a single string (within double quotes). During analysis, these ARXML files are not copied over to a temporary folder but selected from their respective locations.

If you enter the option in an options file (that you later use with the `polyspace-autosar` option `-options-file`), you do not need the double quotes around the string. You can append the `find` string to the `-select-arxml-files` option. For instance, you can enter this line in the options file:

```
-select-arxml-files package -name '*.arxml'
```

File Selection Options

The following `find` options are commonly required for file selection. Use the `i` format of the options for case-insensitive matching.

- `-name`, `-iname`: Match file names with shell patterns.
- `-path`, `-ipath`: Match file paths with shell patterns.
- `-regex`, `-iregex`: Use regular expressions for matching instead of shell patterns.

Use `-not` before an option to exclude files or folders. To specify multiple patterns in a single string, simply place the patterns one after another. For instance, to exclude immediate subfolders `subpackage1` and `subpackage2` from a root folder, use this syntax:

```
-not -path 'subpackage1/*' -not -path 'subpackage2/*'
```

For more information on:

- Shell patterns, see Shell Pattern Matching.
- `find` options, see Finding Files.

File Selection Examples

Some common uses of the file selection options are:

- To specify only the ARXML files beginning with `swc` in the subfolder `sub_package_windowControl` within the root folder `package`, use this syntax:

  ```
  polyspace-autosar -select-arxml-files "package
                       -path 'sub_package_windowControl/*'
                       -name 'swc*.arxml'"
  ```
- To exclude ARXML files from the subfolder `test` at any level in the file structure within the root folder `package`, use this syntax:

  ```
  polyspace-autosar -select-arxml-files "package
                  -not -path '*/test/*'
                  -name '*.arxml'"
  ```
- To exclude ARXML files from the subfolder `test` and subfolder `docs` at any level in the file structure within the root folder `package`, use this syntax:

  ```
  polyspace-autosar -select-arxml-files "package
                 -not -path '*/test/*'
                 -not -path '*/docs/*'
                 -name '*.arxml'"
  ```
- To include all ARXML files from the root folder `package` except those ARXML files containing the string `test`, use this syntax:

  ```
  polyspace-autosar -select-arxml-files "package
                   -name '*.arxml'
                   -not -name '*test*.arxml'"
  ```

Likewise, you can include or exclude `.c` and `.h` files from analysis by using the `-select-source-files` option. Unlike `-select-arxml-files`, this option selects `.c` and `.h` files by default. For instance, to exclude source files from the subfolder `test` at any level in the file structure within the root folder `package`, use this syntax:

```
polyspace-autosar -select-source-files "package -not -path '*/test/*'"
```

Note that with `-select-arxml-files` earlier, you also had to specify the additional pattern `-name '*.arxml'`.

Root Folder Specification

If the root folder name contains spaces, enclose the folder name in double quotes. Because the folder name with file inclusions and exclusions is already in double quotes, you have to escape the additional quotes. For instance, if the root folder is `C:\sdbx\ARXML dir1`, to specify all ARXML files within this folder, use the command:

```
polyspace-autosar -select-arxml-files "\"C:\sdbx\ARXML dir1\" -name '*.arxml'"
```

The additional double quotes around the root folder are escaped as `\"`.

If you specify the option in an options file (that you later use with the `polyspace-autosar` option `-options-file`), you do not need quotes around the `find` string and do not need to escape the additional double quotes. Enter this line in the options file:

```
-select-arxml-files "C:\sdbx\ARXML dir1" -name '*.arxml'
```

See Also

`polyspace-autosar`

More About

- "Run Polyspace on AUTOSAR Code" on page 8-14

Troubleshoot Polyspace Analysis of AUTOSAR Code

This topic describes a component-based approach to verifying AUTOSAR code with Polyspace. For an integration analysis approach, see “Choose Between Component-Based and Integration Analysis of AUTOSAR Code with Polyspace” on page 8-5.

To analyze code implementation of AUTOSAR software components, Polyspace parses the AUTOSAR XML specifications, detects the corresponding code implementation, compiles this code and runs static analysis to detect run-time errors or mismatch between code and specifications. If an error occurs in any of these steps, you do not see analysis results for the software component containing the error. This topic shows how to diagnose and fix these errors.

For sound analysis results, Code Prover requires that your AUTOSAR XML must be well-formed and your code must not have compilation errors. For instance, two elements in your AUTOSAR XML must not have the same Universal Unique Identifier (UUID). You might be using other tools to ensure well-formed ARXML and code without compilation errors. In addition to those tools, you can use the errors during the AUTOSAR XML parsing and code extraction phases of a Code Prover analysis to find issues in your XML and code.

After analysis, open the file `psar_project.xhtml` in a web browser. The file is located in the project folder. Check the overall project status and drill down to the specific software components that have issues. If you create a project in the Polyspace user interface, the **Project Status** tab shows this HTML file after project creation.

View Project Completion Status

If the analysis completes successfully, you see a status message like this.

Project Status

Project is marked created on Sat Dec 23 2017 19:37:53 GMT-0500 (Eastern Standard Time) after completing the following sequence of states in 38.25s:

1. `project_created` entered as created with `no_error` in 0.05s.
2. `project_installed` entered as created with `no_error` in 0.08s.
3. `prove_artifacts_created` entered as created with `no_error` in 1.66s.
4. `user_code_extracted entered` as created with `no_error` in 4.29s.
5. `code_verification_configured` entered as created with `no_error` in 0.2s.
6. `code_verification_executed` entered as created with `no_error` in 31.97s.

In current state, 2 AUTOSAR behaviors are processed, 2 with extracted implementation code and 2 with generated code-prover result.

The message shows how many software components were detected in the ARXML specifications, found in the code implementation and analyzed successfully with Code Prover.

If you create a project in the Polyspace user interface, the analysis is performed later. The project status only shows the first four steps.

View Errors in AUTOSAR XML Parsing

If an error occurs in parsing of AUTOSAR XML (and the error stops the complete analysis), the project status can look like this.

> **Project Status**
>
> Project is marked created on Wed Dec 31 1969 19:25:14 GMT-0500 (Eastern Standard Time) after completing the following sequence of states in 0.58s:
>
> 1. `project_created` entered as created with `no_error` in 0.02s.
> 2. `project_installed` entered as created with `no_error` in 0.09s.
> 3. `prove_artifacts_created` entered as created with `error_in_autosar_prove_artifacts_creation` (2 errors, 0 warnings) in 0.47s.
>
> Execution terminates with `error_in_autosar_prove_artifacts_creation` (2 errors, 1 warnings) See execution log messages

The above message shows that an error occurred when parsing the AUTOSAR XML.

To diagnose further,click the ☰ icon on the upper left. On the left pane, click **Behaviors**. Typically you see the list of all software components whose internal behavior-s are extracted. For each behavior, you can:

- See if there were errors during the ARXML parsing and code extraction phase.
- See further details of errors and warnings.

  Click the link `See key autosar definition for this behavior` for errors and warnings.

For a tutorial on filtering behaviors with errors only and interpreting the errors and warnings, see “Interpret Errors and Warnings in Polyspace Analysis of AUTOSAR Code” on page 8-26.

Tip If you run `polyspace-autosar` at the command-line, you can run only the AUTOSAR XML parsing phase. Fix all errors in your AUTOSAR XML first before continuing the rest of the analysis.

Use the options `-do-not-update-extract-code` and `-do-not-update-verification`.

View Compilation Errors in Code

If a compilation error is found in the source files, the project status can look like this.

Project Status

Project is marked created on Sat Dec 23 2017 19:37:53 GMT-0500 (Eastern Standard Time) after completing the following sequence of states in 38.25s:

1. `project_created` entered as created with `no_error` in 0.05s.
2. `project_installed` entered as created with `no_error` in 0.08s.
3. `prove_artifacts_created` entered as created with `no_error` in 1.66s.
4. `user_code_extracted entered` as created with `error_in_user_code_extraction` (4 errors, 0 warnings) in 4.29s.

Execution terminates with `error_in_user_code_extraction` (4 errors, 1 warnings) See execution log messages

In current state, 2 AUTOSAR behaviors are processed, 2 with extracted implementation code and 2 with generated code-prover result.

The above message shows that an error occurred when extracting the code.

To diagnose further, click the ☰ icon on the upper left. On the left pane, click **Behaviors**. You can see the list of all software components whose internal behavior-s are extracted.

To navigate to the components that have errors, search for the string `error_atLeastOneRunnableInFileThatDoesNotCompile`. Alternatively, to see only the software components with compilation errors, click **Create/Edit Query** in the left pane. Click and deselect the **has success** filter and then click **Search**.

A software component with compilation errors looks like this.

ApplicationComponentBehavior - jyb.tst002.swc001.bhv001

...

...

Extract implementation code

Execution reported no error or warning.

Extraction of implementation completes with state `error_atLeastOneRunnableInFileThatDoesNotCompile`.

Found implementation for 3 of 3 required runnables; extracting 4 files from code-source directory.

Identify which software components have an error. To see the specific error message, click the line that indicates the number of files extracted from code source directory. Click the link **Compiler messages** to open a `.log` file containing all the compilation error messages in the files extracted for the runnable.

The two most common code extraction errors are missing include files and unrecognized data types. For these errors, you can use additional tools to fix many of the errors in one shot. See:

- “Resolve polyspace-autosar Error: Could Not Find Include File” on page 34-35

• "Resolve polyspace-autosar Error: Data Type Not Recognized" on page 34-38

Tip

- If one or more files do not compile, you can still see analysis results for software components where all files passed compilation. In this way, you can analyze certain software components while development is still in progress on the others.
- If you run `polyspace-autosar` at the command-line, you can run only the code extraction phase. Fix all errors in your code first before continuing the analysis.

  Use the options `-do-not-update-autosar-prove-environment` and `-do-not-update-verification`.

See Also

`polyspace-autosar`

More About

- "Run Polyspace on AUTOSAR Code" on page 8-14
- "Resolve polyspace-autosar Error: Conflicting Universal Unique Identifiers (UUIDs)" on page 34-37
- "Resolve polyspace-autosar Error: Could Not Find Include File" on page 34-35
- "Resolve polyspace-autosar Error: Data Type Not Recognized" on page 34-38

Interpret Errors and Warnings in Polyspace Analysis of AUTOSAR Code

This topic describes a component-based approach to verifying AUTOSAR code with Polyspace. For an integration analysis approach, see "Choose Between Component-Based and Integration Analysis of AUTOSAR Code with Polyspace" on page 8-5.

To analyze code implementation of AUTOSAR software components, Polyspace parses the AUTOSAR XML (ARXML) specifications, detects the corresponding code implementation, compiles this code and runs static analysis to detect run-time errors or mismatch between code and specifications. If an error occurs in any of these steps, you do not see analysis results for the software component containing the error. This tutorial shows how to locate and diagnose a class of errors that can occur during parsing of ARXML specifications.

Even if some ARXML specifications have ill-defined elements, the analysis tries to continue further with an ad hoc substitute for those elements, with the assumption that the code implementation might not use those elements. However, for specific types of issues, this substitution is not possible. Therefore, even if the ARXML extraction phase completes with warnings only, the warnings themselves are of two types:

- Warnings that report issues where the analysis is unable to create a substitute.

  For instance, if an event calls a runnable with an undefined port, the analysis cannot model that event.
- Warnings that report issues where the analysis proceeds with a degraded substitute.

  For instance, if a data-type used in a runnable or an RTE API function is undefined, the analysis proceeds with a degraded data type.

To follow the steps in this tutorial, use the demo files in *`polyspaceroot`*`\help\toolbox\codeprover\examples\troubleshooting_polyspace_autosar`.

Overview of File Structure

The demo files consist of a root folder `src` and two options files:

- The options file `options_ko.txt` selects files from the `src` folder that have deliberately introduced errors.
- The options file `options_ok.txt` selects files from the `src` folder that have the same errors fixed.

The folder `src` has two subfolders:

- `arxml` containing the AUTOSAR XML specifications.
- `impl` containing the code implementation of those specifications.

The `arxml` folder has multiple subfolders. Two of these subfolders, `appli` and `interfaces`, have subsubfolders `ok` and `ko` at different levels within the folder structure. The `ok` and `ko` subsubfolders contain the same set of files, except that the files in `ko` have deliberately introduced errors.

See File Selections

Open the options files `options_ok.txt` and `options_ko.txt` in a text editor and note the files selections in each:

- The options file `options_ok.txt` excludes files in `ko` subfolders at any level of the file hierarchy.

  Note the use of the file selection pattern:

  ```
  -not -path '*/ko/*'
  ```
- The options file `options_ko.txt` excludes files in the `ok` subfolders at any level of the file hierarchy..

  Note the use of the file selection pattern:

  ```
  -not -path '*/ok/*'
  ```

In addition, both options files exclude a specific file in the `types` subfolder named `do_not_use_this_arxml_file.arxml` using the pattern:

```
 -path '*/types/*' -not -name 'do_not_use_this_arxml_file.arxml'
```

The full file selection pattern in the file `options_ko.txt` requires that all these criteria must be satisfied:

- The files must not be in a `ko` subfolder at any level of the hierarchy.
- The files must not have the extension `.arxml`.
- The files must be in one of the folders `appli`, `interfaces`, or `types` (except the file `do_not_use_this_arxml_file.arxml`).

For more information on file selection patterns, see “Select AUTOSAR XML (ARXML) and Code Files for Polyspace Analysis” on page 8-19.

Run Analysis

To run the analysis, in a terminal, enter the command:

```
polyspace-autosar -options-file options_ko.txt
```

This command assumes that the path *`polyspaceroot`*`\polyspace\bin` is already added to the `PATH` variable in your operating system. Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\`**`R`**`2023`**`a`**. Otherwise, use the full path to the `polyspace-autosar` command.

Repeat the run with the file `options_ok.txt`.

Note that the options files use the option `-do-not-update-verification` to stop the analysis before the code verification phase.

Interpret Warnings

The results of the analysis with options files `options_ok.txt` and `options_ko.txt` are stored in the folders `project_ok` and `project_ko` respectively.

Navigate to the folder `project_ko` and open the file `psar_project.xhtml` in a web browser. You see errors and warnings both in the ARXML parsing and code extraction phase.

For more information on the errors:

1. Click the ☰ icon on the upper left. On the left pane, click **Behaviors**.

   In the **Detailed status per AUTOSAR Behavior** section, note that:

   - The behavior `tst003.app.swc001.bhv` has warnings in the ARXML parsing phase and errors and warnings in the code extraction phase.
   - The behavior `tst003.app.swc002.bhv` does not have errors or warnings.

   You can also filter out behaviors that do not have errors or warnings. On the left pane, in the **Behavior Selection** section, select **behaviors with error-status** and click **Search**.
2. For further details on the behavior that has errors and warnings, `tst003.app.swc001.bhv`:

   a. In the section **Read AUTOSAR behavior**, click the link **See key autosar definition for this behavior**.
   b. Expand the error message at the top. This error illustrates a situation where Polyspace cannot model an event because of an issue in the ARXML specification. In this case, a port is not defined.
   c. On the left pane, in the **Function selection** section, select **all having error or warning** and click **Search**. You see one other error message. Expand the error message. This error illustrates a situation Polyspace can create a model despite an error. In this case, a data type is not defined and the analysis continues with a degraded type.

   Based on the messages, you can locate the exact errors in the ARXML.

   You also see code extraction errors in this behavior. These code extraction errors can be traced back to the issues in the ARXML. If you fix the issues in the ARXML, the code extraction errors are also fixed. You can see a fixed project by running an analysis with the options file `options_ok.txt`.

See Also

`polyspace-autosar`

More About

- “Run Polyspace on AUTOSAR Code” on page 8-14
- “Troubleshoot Polyspace Analysis of AUTOSAR Code” on page 8-22

Run Polyspace on AUTOSAR Code with Conservative Assumptions

This topic describes a component-based approach to verifying AUTOSAR code with Polyspace. For an integration analysis approach, see “Choose Between Component-Based and Integration Analysis of AUTOSAR Code with Polyspace” on page 8-5.

Polyspace for AUTOSAR runs static program analysis on code implementation of AUTOSAR software components. The analysis looks for possible run-time errors or mismatch with specifications in the AUTOSAR XML (ARXML).

The default analysis assumes that pointer arguments to runnables and pointers returned from `Rte_` functions are not NULL. For instance, in this example, the analysis assumes that `aInput`, `aOutput` and `aOut2` are not NULL. The conditions that compare these arguments against `NULL_PTR` always evaluate to false and appear gray in the results. Here, `NULL_PTR` is a macro that represents NULL.

```
iOperations_ApplicationError foo(
    Rte_Instance const self,
    app_Array_2_n320to320ConstRef aInput,
    app_Array_2_n320to320Ref aOutput,
    app_Enum001Ref aOut2)
{
    iOperations_ApplicationError rc = E_NOT_OK;
    if (aInput==NULL_PTR) {
        rc = RTE_E_iOperations_ERR001;
    } else if (aOutput==NULL_PTR) {
        rc = 43;
    } else {
        unsigned int i=0;
        for (;i<2U;++i) {
            aOutput[1-i] = aInput[i];
        }
        if (aOut2!=NULL_PTR) {
            *aOut2 = 1234;
            rc = RTE_E_OK;
        }
    }
    return rc;
}
```

You might want to run a conservative analysis where pointer arguments to runnables and pointers returned from `Rte_` functions can be NULL-valued. The conservative analysis helps you determine if you have guarded against the possibility of NULL-valued pointers within your runnable.

To allow the possibility of NULL-valued pointers from external sources, undefine the macro `RTE_PTR2USERCODE_SAFE`. To undefine a macro, use one of these methods depending on how you run the analysis.

- In the Polyspace user interface, the macro is defined with the option `Preprocessor definitions (-D)`. Remove the macro from this option and move to the option `Disabled preprocessor definitions (-U)`.
- If you run `polyspace-autosar` at the command-line, use the option `-U` to undefine the macro.

If you disable the macro, you no longer see unreachable code when comparing pointers arguments to runnables against NULL. To see the effect of this macro, run a conservative Polyspace analysis on the demo files in `*polyspaceroot*\help\toolbox\codeprover\examples\polyspace_autosar`.

See Also

`polyspace-autosar`

More About

- "Run Polyspace on AUTOSAR Code" on page 8-14

Run Polyspace on AUTOSAR Code Using Build Command

This topic describes a component-based approach to verifying AUTOSAR code with Polyspace. For an integration analysis approach, see "Choose Between Component-Based and Integration Analysis of AUTOSAR Code with Polyspace" on page 8-5.

If you use the AUTOSAR methodology for software development with a build command for compilation, you can reuse existing artifacts to specify source files and compilation options for Code Prover.

- You can reuse the source file specification in AUTOSAR XML files.

  Polyspace can read AUTOSAR XML specifications and extract source files involved in each software component into modules for subsequent Code Prover run-time checks. If you use the AUTOSAR methodology for software development, you can reuse the modularization built into this methodology for a modular Code Prover analysis. See `polyspace-autosar`.
- You can reuse compilation options specified in your build command.

  Polyspace can trace your build command and detect the compiler invoked along with compilation options such as paths to standard includes and macro definitions. See `polyspace-configure`.

This topic shows how to combine the two approaches and automate your Code Prover analysis.

To follow the steps in this tutorial, use the demo files in *polyspaceroot*`/help/toolbox/codeprover/examples/polyspace_autosar_configure`.

The example uses a Linux-based makefile and Linux path separators. To run the example in Windows, make appropriate modifications.

Run Code Prover Without Compilation Options

Copy the contents of the demo folder into a temporary folder, for instance, `/tmp/demo/`. Navigate to that folder.

```
cd /tmp/demo
```

Run Code Prover on the ARXML and source files in the subfolder `mRtwDemoAutosar_autosar_rtw`. Save results in the folder `/tmp/res`.

```
polyspace-autosar -create-project /tmp/res \
-arxml-dir mRtwDemoAutosar_autosar_rtw \
-sources-dir mRtwDemoAutosar_autosar_rtw
```

Note the compilation errors. For instance, in the `/tmp/res/.extract` folder, open the file `GPIO_read.log`. You see a `#error` directive because the macro `MY_DEFINE_FROM_SIMULINK` is not defined.

If you open the file `GPIO_read.c` in `/tmp/demo/mRtwDemoAutosar_autosar_rtw`, you see the line causing the issue.

```
#ifndef MY_DEFINE_FROM_SIMULINK
#error Missing MY_DEFINE_FROM_SIMULINK
#endif
```

This line is supposed to cause an error during preprocessing unless the macro `MY_DEFINE_FROM_SIMULINK` is defined.

Run Code Prover with Compilation Options from Build Command

The makefile `mRtwDemoAutosar.mk` in `/tmp/demo/mRtwDemoAutosar_autosar_rtw` defines macros and paths to include folders. For instance, the previously missing macro `MY_DEFINE_FROM_SIMULINK` is defined in the line:

```
DEFINES_CUSTOM = -DMY_DEFINE_FROM_SIMULINK
```

Navigate to the folder containing the makefile.

```
cd /tmp/demo/mRtwDemoAutosar_autosar_rtw
```

Extract compilation options from a build command that uses this makefile `mRtwDemoAutosar.mk`. For instance, if you installed MATLAB in `/usr/local/MATLAB/R2018b`, you can trace the makefile like this.

```
polyspace-configure -no-sources \
-output-options-file psoptions -allow-overwrite\
make -B -f mRtwDemoAutosar.mk START_DIR=.. \
MATLAB_ROOT=/usr/local/MATLAB/R2018b buildobj
```

The compilation options in the makefile are converted to Polyspace analysis options and saved in the options file `psoptions`. The `-no-sources` option ensures that the `polyspace-configure` command extracts compilation options only and not sources. `START_DIR` and `MATLAB_ROOT` are variables specific to the demo makefile and might not be required in other makefiles that you use.

Remove results from any previous run of the `polyspace-autosar` command.

```
rm -r /tmp/res
```

Provide the options file `psoptions` created in the previous step to the `polyspace-autosar` command.

```
polyspace-autosar -create-project /tmp/res \
-arxml-dir . \
-sources-dir .\
-extra-options-file psoptions
```

You no longer see the compilation errors because Code Prover is now aware of the compilation options that you used in your build command.

See Also

`polyspace-autosar`

More About

- "Run Polyspace on AUTOSAR Code" on page 8-14

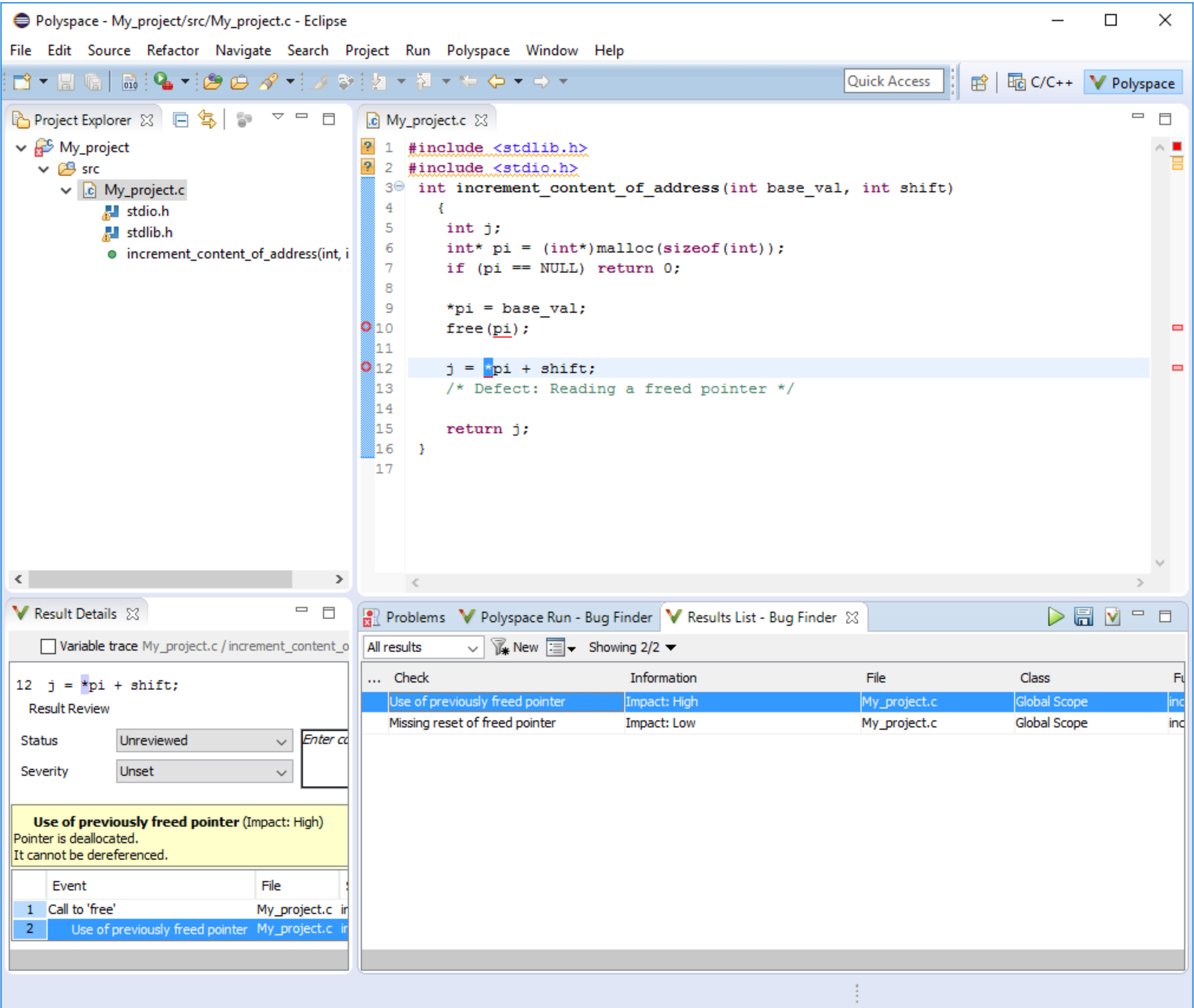
Run Polyspace Analysis in IDE Plugins

Run Polyspace Analysis on Eclipse Projects

This topic describes how to run a Polyspace analysis on complete Eclipse projects using Polyspace Bug Finder or Polyspace Code Prover. For the Polyspace as You Code plugin, see “Run Polyspace as You Code in Eclipse and Review Results”.

If you develop code in Eclipse or an Eclipse-based IDE, you can install the Polyspace plugin and run a Polyspace analysis on the source files in an Eclipse project. You can check for bugs each time you save your code, or explicitly run an analysis.

This topic describes how to set up a Polyspace analysis in Eclipse and review analysis results.



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

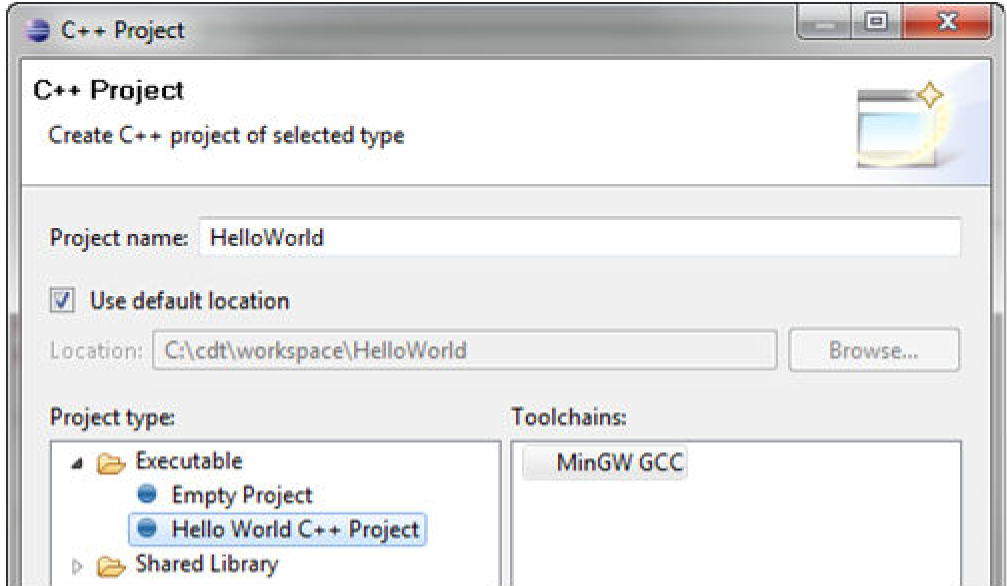
The analysis progress bar, quick run buttons and analysis results appear on specific panes. To avoid cluttering your window, you can confine these panes to the Polyspace perspective. Select **Window > Open Perspective > Other**. In the Open Perspective dialog box, select **Polyspace**. You can switch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

Configure and Run Analysis

Configure analysis

Polyspace analyzes the source files in your Eclipse project. In addition to sources, the analysis uses the following information:

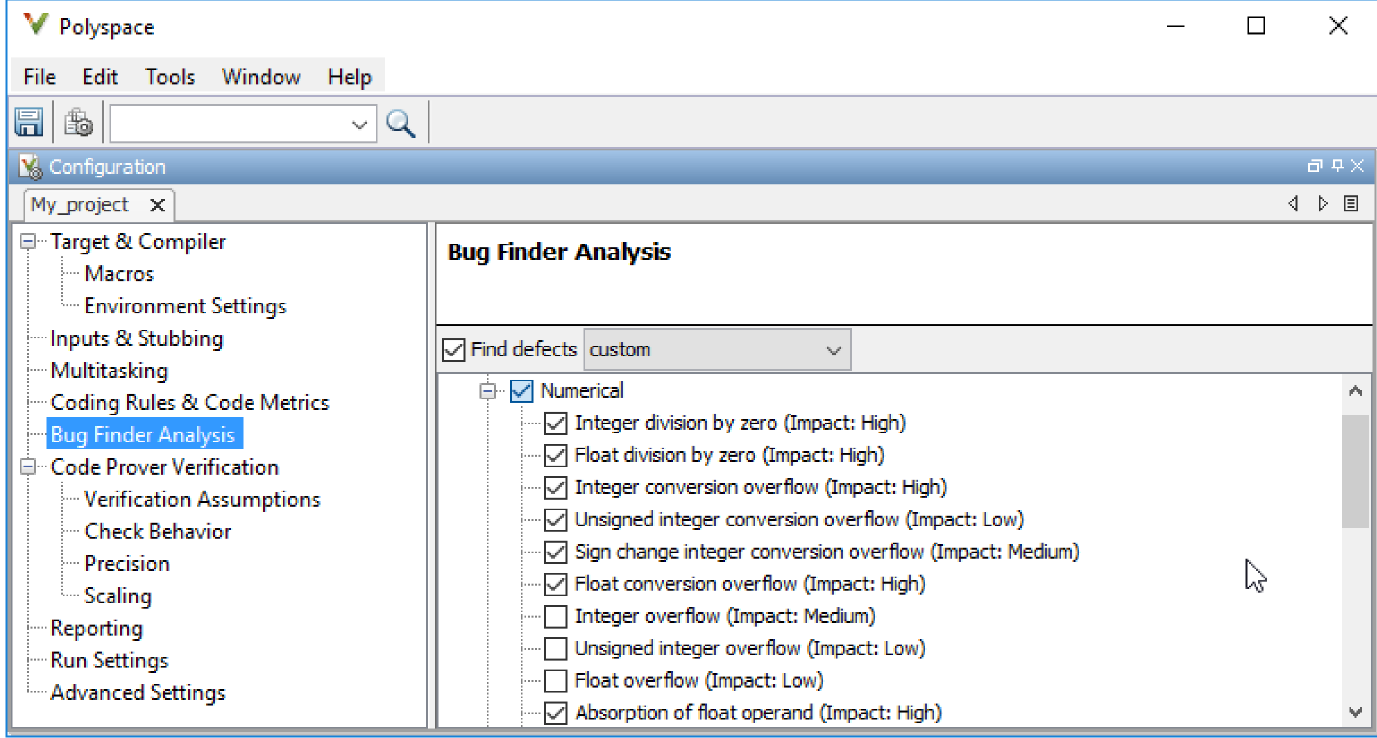
- Compiler: The compiler toolchain can be extracted from your Eclipse project. If the project directly refers to a compilation toolchain such as MinGW GCC, the Polyspace analysis can use the information.

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

  If Polyspace cannot extract the compiler information from your build command, you can also explicitly specify your compiler options explicitly like other analysis options.

  See “Specify Polyspace Compiler Options Through Eclipse Project” on page 9-7.
- 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 “Code Prover Analysis on Clusters”.

Run analysis

After configuration, you can start and stop a Polyspace analysis explicitly from the **Polyspace** menu, right-click options on your Eclipse project or quick run buttons in the Polyspace panes. You can switch between Bug Finder and Code Prover using the icon on the **Polyspace Run** pane.

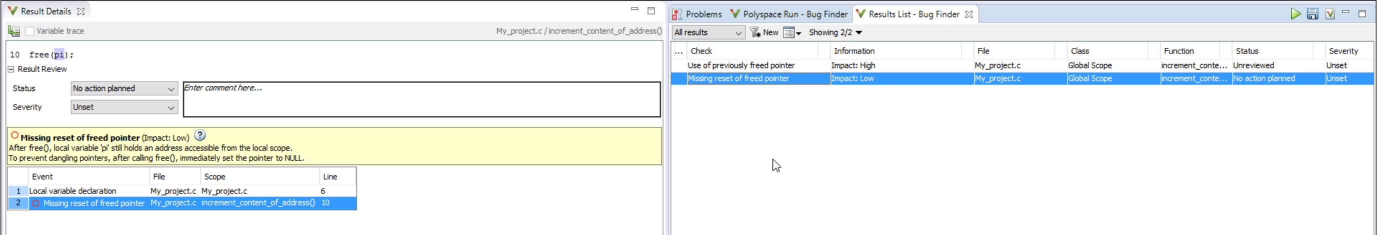
Run analysis when saving code

In the Polyspace perspective, you can set up a Bug Finder analysis that runs each time you save your code. To enable this analysis, select **Polyspace** > **Run Fast Analysis on Save**. The analysis runs quickly but looks for a reduced set of defects. You get the same results as if you had specified the analysis option `Use fast analysis mode for Bug Finder (-fast-analysis)`.

Review Analysis Results

View results after analysis

After analysis, the results appear on the **Results List** pane. Click each result to see the source code and details on the **Result Details** pane.



View results as available

Some results of a Bug Finder analysis are often available before the analysis is complete. If so, the icon in the **Polyspace Run - Bug Finder** pane turns to . To load available results, click this icon. The icon shows up again when more results are available.

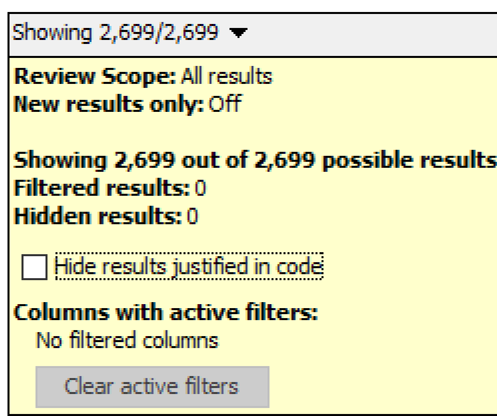
Address results

Based on the result details, fix your code or justify the result. To justify a result, set its **Status** to `Justified`, `No Action Planned` or `Not a Defect`. To hide a justified result in the next run, add the status as annotation to your source code. See “Annotate Code and Hide Known or Acceptable Results” on page 30-2.

For quick annotation, right-click the result and select **Annotate Code and Hide Result**. The option adds annotations in this format and hides the result from the results list:

```
line of code; /* polyspace Family:Result_name */
```

For details of the format, see “Annotate Code and Hide Known or Acceptable Results” on page 30-2. To unhide the hidden results, from the **Showing** menu, clear the box **Hide results justified in code**.



See Also

Related Examples

- “Specify Polyspace Compiler Options Through Eclipse Project” on page 9-7
- “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2
- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2
- “Filter and Group Results in Polyspace Desktop User Interface” on page 24-2

Specify Polyspace Compiler Options Through Eclipse Project

This topic describes how to configure a Polyspace analysis of Eclipse projects using Polyspace Bug Finder or Polyspace Code Prover. For the Polyspace as You Code plugin, see “Run Polyspace as You Code in Eclipse and Review Results”.

Polyspace analysis in Eclipse uses a set of default analysis options preconfigured for your coding language and operating system. For each project, you can customize the analysis options further.

- Compiler options: You specify the compiler that you use, the libraries that you include and the macros that are defined for your compilation.
  - If your Eclipse project directly refers to a compilation toolchain, the analysis reads the compiler options from the project.

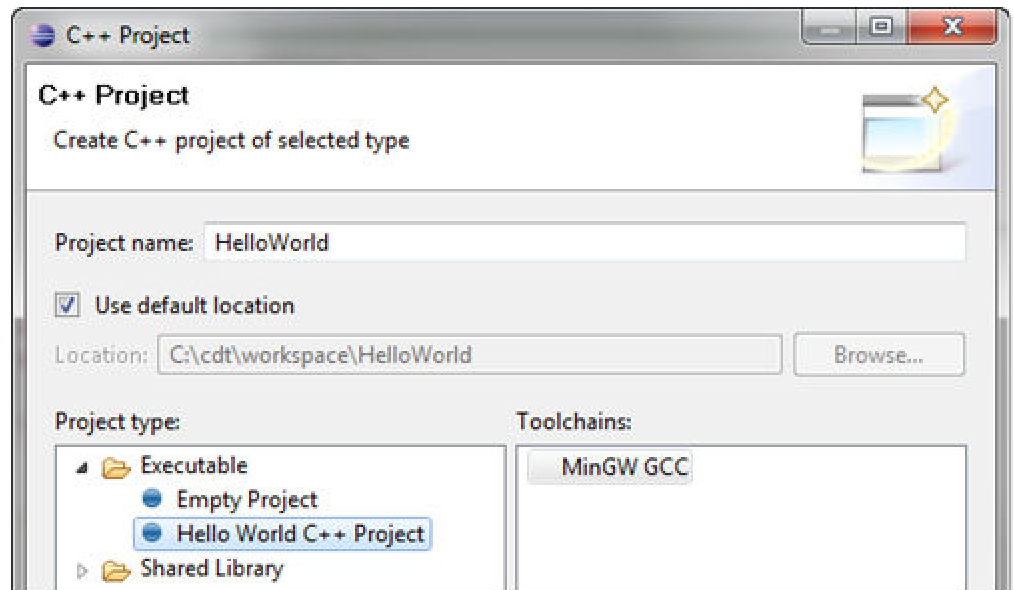
    See “Eclipse Refers Directly to Your Compilation Toolchain” on page 9-7.
  - If the project refers to your compilation toolchain through a build command, the analysis cannot read the compiler options directly. Trace the build command to extract the options. Tracing a build command involves first executing the command and extracting required information from the processes executed.

    See “Eclipse Uses Your Compilation Toolchain Through Build Command” on page 9-8.
- 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 9-2.

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:

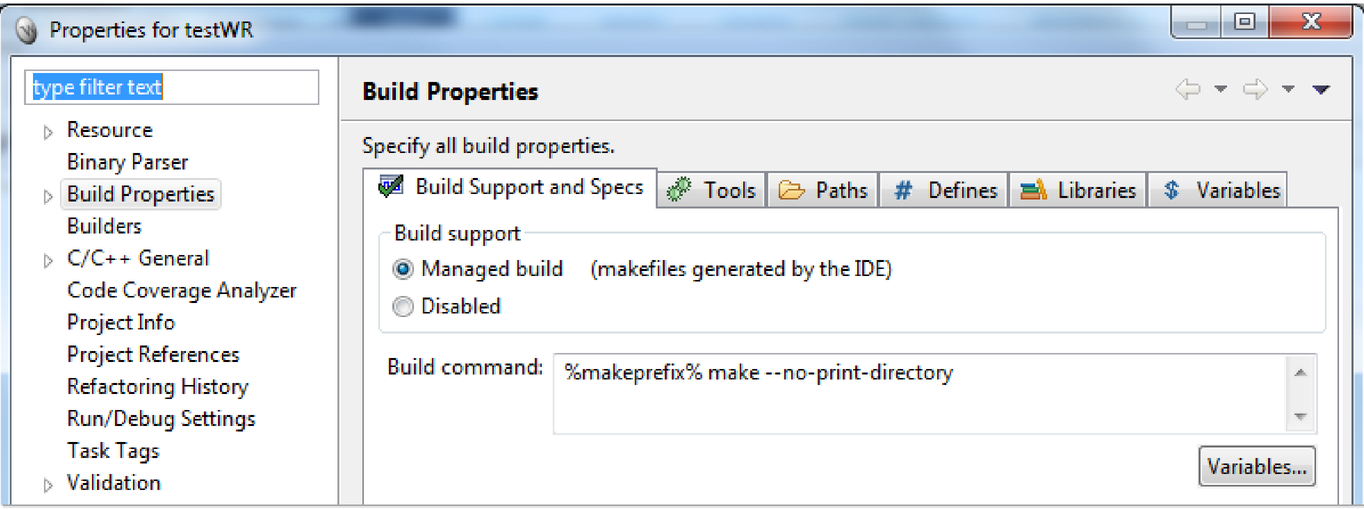
- See the include paths under the **Includes** tab.

  During analysis, the paths are implicitly used with the analysis option `Include folders (-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 9-2

Configure Analysis on Server

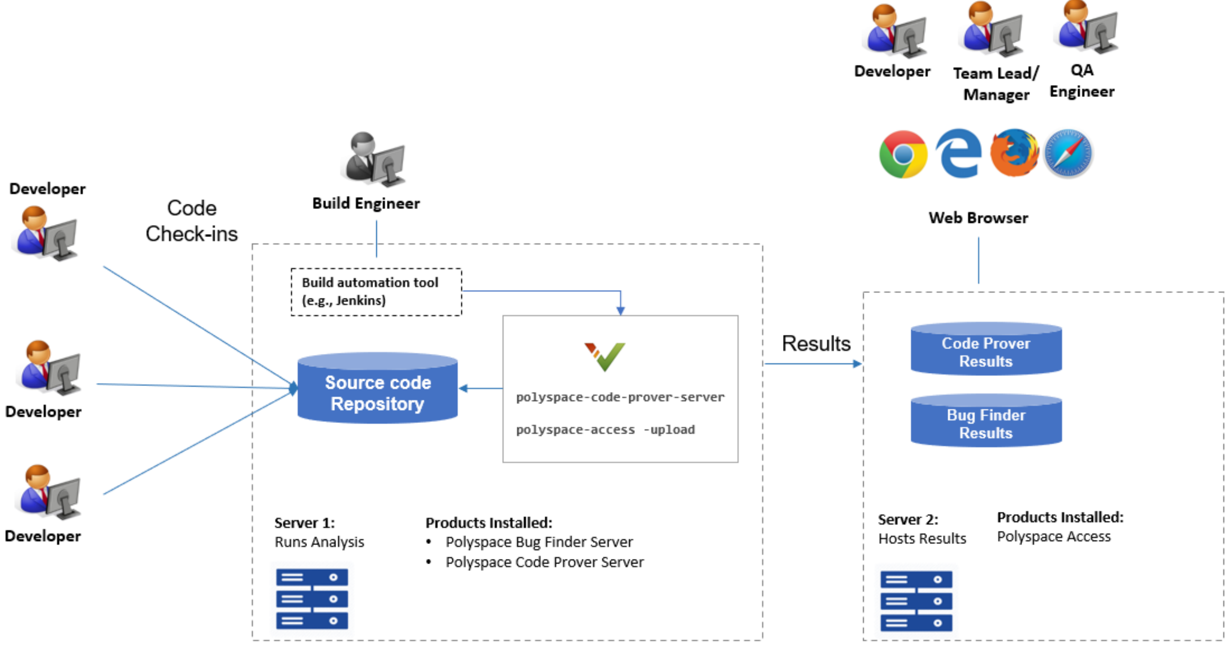
Run Polyspace Analysis on Servers

Run Polyspace Code Prover on Server and Upload Results to Web Interface

Polyspace Code Prover Server proves the absence of run-time errors in C/C++ code, and then uploads findings to a web interface for code review.

You can run Code Prover as part of continuous integration. Set up scripts that run a Code Prover analysis at regular intervals or based on new code submissions. The scripts can upload the analysis results for review in the Polyspace web interface and optionally send emails to owners of source files with Polyspace findings. The owners can open the web interface to review only the new findings from their submission, and then fix or justify the issues.

In a typical project or team, Polyspace Code Prover Server runs periodically on a few testing servers and uploads the results for review. Each developer and quality engineer in the team has a Polyspace Code Prover Access license to view the results in the web interface for investigation and bug fixing.



Prerequisites

To run a Code Prover analysis on a server and review the results in the Polyspace Code Prover Access web interface, perform this one-time setup:

- To run the analysis, install one instance of the Polyspace Code Prover Server product.
- To upload results, set up the components required to host the web interface of Polyspace Code Prover Access.
- To view the uploaded results, you and each developer reviewing the results must have a Polyspace Code Prover Access license.

See “Install Polyspace Server and Access Products”.

Check Polyspace Installation

To check if Polyspace Code Prover Server is installed:

1. Open a command window. Navigate to *polyspaceserverroot*\polyspace\bin. Here, *polyspaceserverroot* is the Polyspace Code Prover Server installation folder, for instance, C:\Program Files\Polyspace Server\R2023a. See also “Installation Folder”.
2. Enter:

   ```
   polyspace-code-prover-server -help
   ```

You should see the list of options allowed for a Code Prover analysis.

To check if the Polyspace web interface is set up for upload:

1. Navigate again to *polyspaceserverroot*\polyspace\bin.
2. Enter:

   ```
   polyspace-access -host hostName -port portNumber -create-project testProject
   ```

   Here, *hostName* is the name of the server hosting the Polyspace Code Prover Access web server. For a locally hosted server, use localhost. *portNumber*I is the optional port number of the server. If you omit the port number, 9443 is used.

   If the setup was complete, a project called testProject is created in the Polyspace web interface.

   You are prompted for your login and password each time you use the polyspace-access command. To avoid entering login information each time, provide the login and an encrypted version of your password with the command. To create an encrypted password, enter:

   ```
   polyspace-access -encrypt-password
   ```

   Enter your login and password. Copy the encrypted password and provide this encrypted password with the -encrypted-password option when using the polyspace-access command.
3. In a web browser, open this URL:

   https://*hostName*:*portNumber*/metrics/index.html

   Here, *hostName* and *portNumber* are the host name and port number from the previous step.

In the **Project Explorer** pane on the Polyspace web interface, you should see the newly created project testProject.

Run Code Prover on Sample Files

To run Code Prover, in your operating system, open a command window.

1. To run a Code Prover analysis, use the polyspace-code-prover-server command.

2 To upload the results to the Polyspace web interface, use the `polyspace-access` command.

To avoid typing the full path to the command, add the path *`polyspaceserverroot`*`\polyspace\bin` to the `Path` environment variable on your operating system.

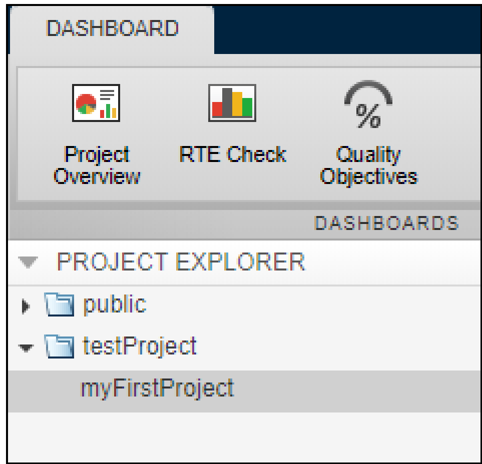
Try out the commands on sample files provided with your Polyspace installation.

1 Copy the sample source files from *`polyspaceserverroot`*`\polyspace\examples\cxx\Code_Prover_Example\sources` to another folder where you have write permissions. Navigate to this folder at the command line.
2 Enter:

```
polyspace-code-prover-server -sources example.c,single_file_analysis.c
                -I . -main-generator -results-dir .
polyspace-access -host hostName -port portNumber
                -login username -encrypted-password pwd
                -create-project testProject
polyspace-access -host hostName -port portNumber
                -login username -encrypted-password pwd
                -upload . -project myFirstProject -parent-project testProject
```

Here, *`username`* is your login name and *`pwd`* is the encrypted password that you created previously. See "Check Polyspace Installation" on page 10-3.

Refresh the Polyspace web interface. You see the newly uploaded results under the `testProject` folder in the **Project Explorer** pane.



To see the results in the project, click **Review**. For more information, see "Review Polyspace Code Prover Results in Web Browser". You can also access the documentation using the button in the upper right of the Polyspace Access interface.

The options used with the `polyspace-code-prover-server` command are:

- `-sources`: Specify comma-separated source files.
- `-I`: Specify path to include folder. Use the `-I` flag each time you want to add a separate include folder.
- `-results-dir`: Specify the path to the folder where Polyspace Code Prover results will be saved.

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

- `Verify module or library (-main-generator)`: Specify that a `main` function must be generated if not found in the source files

For the full list of options available for a Code Prover analysis, see "Complete List of Polyspace Code Prover Analysis Options". To open the Code Prover documentation in a help browser, enter:

```
polyspace-code-prover-server -doc
```

Sample Scripts for Code Prover Analysis on Servers

To run the analysis, instead of typing the commands at the command line, you can use scripts. The scripts can execute each time that you add or modify source files.

A sample Windows batch file is shown below. Here, the path to the Polyspace installation is specified in the script. To use this script, replace `polyspaceserverroot` with the path to your installation. You must have already generated the encrypted password for use in the scripts. See "Check Polyspace Installation" on page 10-3.

```
echo off
set POLYSPACE_PATH=polyspaceserverroot\polyspace\bin
set LOGIN=-host hostName -port portNumber -login username -encrypted-password pwd
"%POLYSPACE_PATH%\polyspace-code-prover-server"
  -sources example.c,single_file_analysis.c -I .^
  -main-generator -results-dir .
"%POLYSPACE_PATH%\polyspace-access" %LOGIN% -create-project testProject
"%POLYSPACE_PATH%\polyspace-access" %LOGIN% -upload . -project myFirstProject
  -parent-project testProject
pause
```

A sample Linux shell script is shown below.

```
echo off
set POLYSPACE_PATH=polyspaceserverroot\polyspace\bin
set LOGIN=-host hostName -port portNumber -login username -encrypted-password pwd
"%POLYSPACE_PATH%\polyspace-code-prover-server"
  -sources example.c,single_file_analysis.c -I .^
  -main-generator -results-dir .
"%POLYSPACE_PATH%\polyspace-access" %LOGIN% -create-project testProject
"%POLYSPACE_PATH%\polyspace-access" %LOGIN% -upload . -project myFirstProject
  -parent-project testProject
pause
```

Specify Sources and Options in Separate Files from Launching Scripts

Instead of listing the source files and analysis options within the launching scripts, you can list them in separate text files.

- Specify the text file listing the sources by using the option `-sources-list-file`.
- Specify the text file listing the analysis options by using the option `-options-file`.

By removing the source files and analysis option specifications from the launching scripts, you can modify these specifications as required with new code submissions while leaving the launching script untouched.

Consider the script in the preceding example. You can modify the `polyspace-code-prover-server` command to use text files with sources and options. Instead of:

```
polyspace-code-prover-server -sources example.c,single_file_analysis.c -I .
   -main-generator -results-dir .
```

use:

```
polyspace-code-prover-server -sources-list-file sources.txt
   -options-file polyspace_opts.txt -results-dir .
```

Here:

- `sources.txt` lists the source files:

  ```
  example.c
  single_file_analysis.c
  ```
- `polyspace_opts.txt` lists the analysis options in separate lines:

  ```
  -I .
  -main-generator
  ```

Typically, your source files are specified in a build command (makefile). Instead of specifying the source files directly, you can trace the build command to create a list of source specifications. See `polyspace-configure`.

Complete Workflow

In a typical continuous integration workflow, you run a script that executes these steps:

1. Extract Polyspace options from your build command.

   For instance, if you use makefiles to build your source code, you can extract analysis options from the makefile.

   ```
   polyspace-configure -output-options-file compile_opts make
   ```

   See also:

   - `polyspace-configure`
   - “Create Polyspace Analysis Configuration from Build Command (Makefile)” on page 13-21
2. Run the analysis with the previously created options file. Append a second options file that contains the remaining options required for the analysis.

   ```
   polyspace-code-prover-server -options-file compile_opts -options-file run_opts
   ```
3. Upload the results to Polyspace Code Prover Access.

   ```
   polyspace-access login -upload resultsFolder -project projName
     -parent-project parentProjName
   ```

Here, *login* is the combination of options required to communicate with the web server that is hosting Polyspace Code Prover Access:

```
-host hostName -port portNumber -login username -encrypted-password pwd
```

resultsFolder is the folder containing the Polyspace results. *projName* and *parentProjName* are names of the project and parent folder as they would appear in the Polyspace Code Prover Access web interface.

4. Optionally, send email notifications to developers with new results from their code submission. The email contains attachments with links to the results in the Polyspace Code Prover Access web interface.

   See "Send Email Notifications with Polyspace Code Prover Server Results".

See examples of scripts executing these steps in "Sample Scripts for Polyspace Analysis with Jenkins" on page 10-17.

See Also

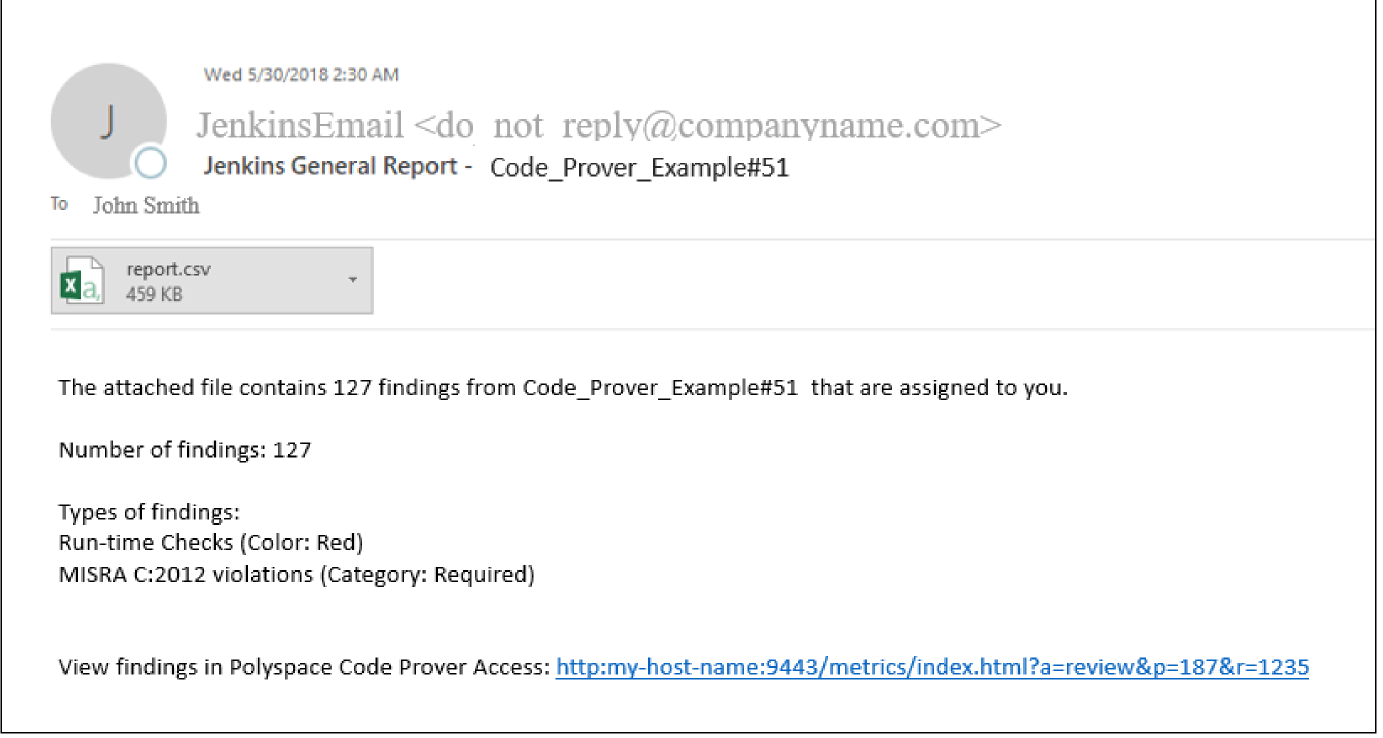
`polyspace-access` | `polyspace-code-prover-server`

More About

- "Send Email Notifications with Polyspace Code Prover Server Results"
- "Complete List of Polyspace Code Prover Analysis Options"

Send Email Notifications with Polyspace Code Prover Server Results

If you run a Polyspace analysis as part of continuous integration, each new code submission produces new results. You not only see new results in components that were modified but also in components that depended on the modified components. You can set up e-mail alerts so that component owners get notified when new Polyspace results appear in their components.



Creating E-mail Notifications

To create e-mail notifications:

1. Export new analysis results to a tab-delimited text file (`.tsv` format).

   Apply filters to export specific types of results, for instance, defects with high impact. If required, you can also apply additional filters to the exported files using search and replace utilities. See “Export Results for E-mail Attachments” on page 10-10.
2. Send an email with the results file in attachment. For each result, the attachment contains links to open the result in the Polyspace Code Prover Access web interface.

   For instance, if you use an e-mail plugin in Jenkins, you can create a post-build step to send an e-mail after the analysis is complete.

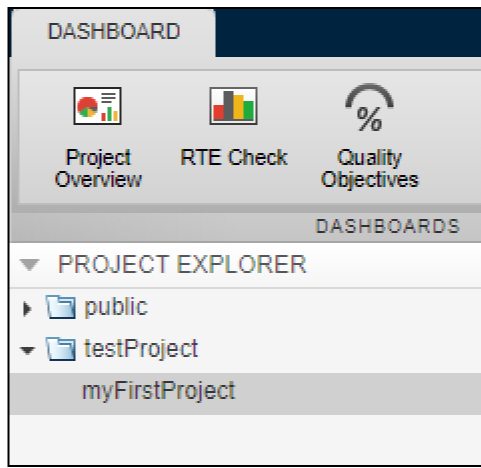
If you use the Polyspace plugin in Jenkins, you can use Polyspace helper utilities for the entire e-mail notification process. See “Sample Scripts for Polyspace Analysis with Jenkins” on page 10-17.

Alternatively, results can be directly assigned to owners based on their file paths. You can set up email notifications that exports a separate results file per owner and sends an email to each owner with the corresponding results file in attachment.

Prerequisites

To run this tutorial:

- You must have uploaded some result in the Polyspace Code Prover Access interface. If you complete the tutorial “Run Polyspace Code Prover on Server and Upload Results to Web Interface”, you should see a folder `testProject` on the **Project Explorer** pane. The folder contains one project `myFirstProject`.

  To see the results in the project, with `myFirstProject` selected, click the **Review** button. You see a list of run-time checks. The **Type** column shows the color of the checks. In this tutorial, only red checks will be exported for e-mail attachments.

- You must be able to interact with the Polyspace Code Prover Access interface from the command line. For instance, navigate to *`polyspaceserverroot`*`\polyspace\bin` and enter:

  ```
  polyspace-access login -list-project
  ```

  Here. *`polyspaceserverroot`* is the Polyspace Code Prover Server installation folder, for instance, `C:\Program Files\Polyspace Server\R2023a`.The variable *`login`* refers to the following combination of options. You provide these options with every use of the `polyspace-access` command.

  ```
  -host hostName -port portNumber -login username -encrypted-password pwd
  ```

  Here, *`hostName`* is the name of the Polyspace Code Prover Access web server. For a locally hosted server, use `localhost`. *`portNumber`* is the optional port number of the server. If you omit the port number, `9443` is used. *`username`* and *`pwd`* refer to the login and an encrypted version of your password. To create an encrypted password, enter:

  ```
  polyspace-access -encrypt-password
  ```

  Copy the encrypted password and provide this password with later uses of the `polyspace-access` command.

Export Results for E-mail Attachments

You can export all results in a project or only certain types of results.

Open a command window. Navigate to the folder where you want to export the results.

- To export all results, enter the following:

```
polyspace-access login -export testProject/myFirstProject -output .\result.txt
```

- To export only red checks, enter the following:

```
polyspace-access login -export testProject/myFirstProject
     -rte Red -output .\result_red_checks.txt
```

Open each text file in a spreadsheet viewing utility such as Microsoft® Excel®. In the first file, you see all results but in the second file, you only see the red run-time checks. Instead of `-rte Red`, you can apply other filters.

- To see only new results compared to the previous analysis of the same project, use the option `-new-findings`.
- To apply a more fine-grained set of filters, you can use software quality objectives (SQOs). The software quality objectives are specified through a progressively stricter set of SQO levels, numbered from 1 to 6. You can customize the requirements of each level in the Polyspace Access web interface, and then use the option `-open-findings-for-sqo` with the level number to export only those results that must be reviewed to meet the requirements. See also "Evaluate Polyspace Code Prover Results Against Software Quality Objectives" on page 31-2.

To see all filtering options, enter:

```
polyspace-access -h -export
```

You can configure your e-mail utility to send these exported files in attachment.

If required, you can also apply additional filters to the exported files using search and replace utilities. For instance, use search and replace utilities on the results file to include results only from specific files and functions. In Linux, you can use `grep` and `sed` to retain only results in specific files.

Assign Owners and Export Assigned Results

You can assign owners to results in specific files or folders. You can then export one result file per owner and send an email to each owner with the corresponding file in attachment.

You can assign owners in the Polyspace Code Prover Access web interface or at the command line.

In this tutorial, assign all results in the file `example.c` to `jsmith` and all results in the file `single_file_analysis.c` to `jboyd`.

```
polyspace-access login
     -set-unassigned-findings testProject/myFirstProject
     -owner jsmith -source-contains example.c
polyspace-access login
     -set-unassigned-findings testProject/myFirstProject
     -owner jboyd -source-contains single_file_analysis.c
```

After assignment, export one results file per owner.

```
polyspace-access -host login
    -export testProject/myFirstProject -output .\results.txt -output-per-owner
```

These files contain the exported results:

- `results.txt` contains all results.
- `results_jsmith.txt` and `results_jboyd.txt` contains results assigned to `jsmith` and `jboyd` respectively.
- `results.txt.owners.list` contains the list of owners, in this case:

  ```
  jsmith
  jboyd
  ```

Before assigning owners to results, use the option `-dryrun` to perform a dry run of the assignments. Without performing the assignment, the option shows the files with results that are assigned and the owner that the results are assigned to.

See Also

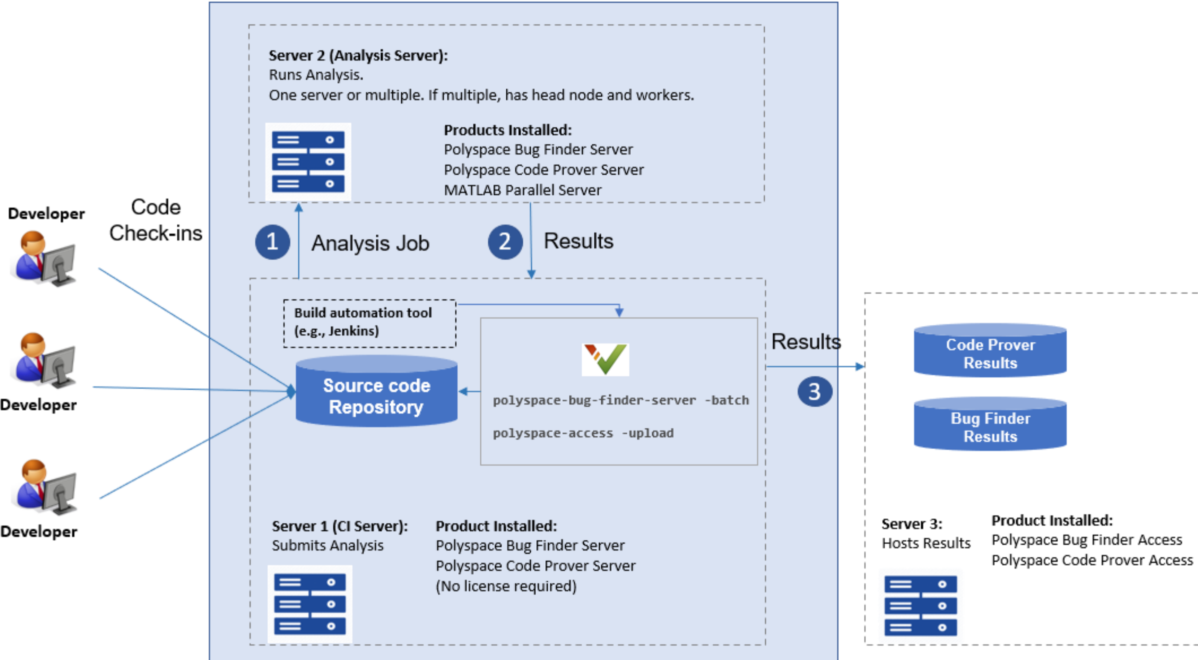
`polyspace-access`

Offload Polyspace Analysis from Continuous Integration Server to Another Server

When running static code analysis with Polyspace as part of continuous integration, you might want the analysis to run on a server that is different from the server running your continuous integration (CI) scripts. For instance:

- You might want to perform the analysis on a server that has more processing power. You can offload the analysis from your CI server to the other server.
- You might want to submit analysis jobs from several CI servers to a dedicated analysis server, hold the jobs in queue, and execute them as Polyspace Server instances become available.

When you offload an analysis, the compilation phase of the analysis runs on the CI server. After compilation, the analysis job is submitted to the other server and continues on this server. On completion, the analysis results are downloaded back to the CI server. You can then upload the results to Polyspace Access for review, or report the results in some other format.



Install Products

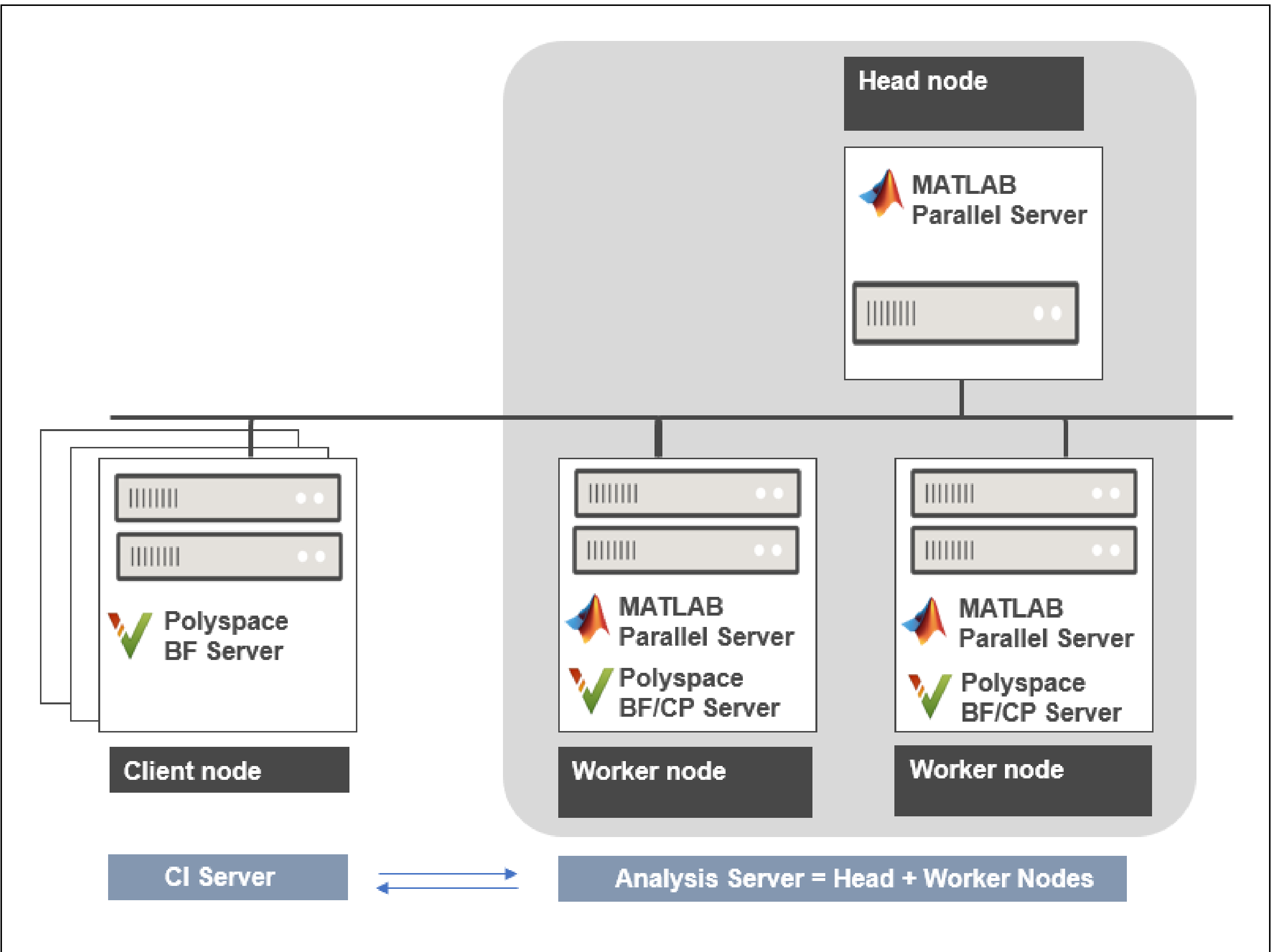
A typical distributed network for offloading an analysis consists of these parts:

- **Client node(s)**: Each CI server acts as a client node that submits Polyspace analysis jobs to a cluster.

  The cluster consists of a head node and one or more worker nodes. In this example, we use the same computer as the head node and one worker node.

- **Head node**: The head node distributes the submitted jobs to worker nodes.
- **Worker node(s)**: Each worker node executes one Polyspace analysis at a time.

Note The versions of Polyspace on the client and worker nodes must match.



Install these products:

- **Client nodes**: Polyspace Bug Finder Server or Polyspace Code Prover Server to submit jobs from the Continuous Integration server. Note that you do not require licenses for the Polyspace Server products if you use them only for job submission (with the `-batch` option).
- **Head node**: MATLAB Parallel Server™ to manage submissions from multiple clients. An analysis job is created for each submission and placed in a queue. As soon as a worker node is available, the next analysis job from the queue is run on the worker.
- **Worker node(s)**: MATLAB Parallel Server and Polyspace Bug Finder Server or Polyspace Code Prover Server on the worker nodes to run a Bug Finder or Code Prover analysis.

In the simplest configuration, where the same computer serves as the head node and one worker node, you install MATLAB Parallel Server and one or both Polyspace Bug Finder Server and Polyspace

Code Prover Server on this computer. This example describes the simple configuration but you can generalize the steps to multiple workers on separate computers.

Configure and Start Job Scheduler Services on Head Node and Worker Node

Start a job scheduler service (the MATLAB Job Scheduler or `mjs` service) on the computer that acts as the head node and worker node. Before starting the service, you must perform an initial setup.

Specify Polyspace Installation Paths

MATLAB Parallel Server and Polyspace Server products are installed in two separate folders. The MATLAB Parallel Server installation routes the Polyspace analysis to the Polyspace Server products. To link the two installations, specify the path to the root folder of the Polyspace Server products in your MATLAB Parallel Server installation.

1. Navigate to `*matlabroot*\toolbox\parallel\bin\`. Here, *`matlabroot`* is the MATLAB Parallel Server installation folder, for instance, `C:\Program Files\MATLAB\R2023a`.
2. Uncomment and modify the following line in the file `mjs_polyspace.conf`. To edit and save the file, open your editor in administrator mode.

   ```
   POLYSPACE_SERVER_ROOT=polyspaceserverroot
   ```

   Here, *`polyspaceserverroot`* is the installation path of the server products, for instance:

   ```
   C:\Program Files\Polyspace Server\R2023a
   ```

The Polyspace Server product offloading the analysis must belong to the same release as the Polyspace Server product running the analysis. If you offload an analysis from an R2023a Polyspace Server product, the analysis must run using another R2023a Polyspace Server product.

Configure mjs Service Settings

Before starting MATLAB Parallel Server (the `mjs` service), you must perform a minimum configuration.

1. Navigate to `*matlabroot*\toolbox\parallel\bin`, where *`matlabroot`* is the MATLAB Parallel Server installation folder, for instance, `C:\Program Files\MATLAB\R2023a`.
2. Modify the file `mjs_def.bat` (Windows) or `mjs_def.sh` (Linux). To edit and save the file, open your editor in administrator mode.

   Read the instructions in the file and uncomment the lines as needed. At a minimum, uncomment these lines that specify:

   - Host name.

     Windows:

     ```
     REM set HOSTNAME=%strHostname%.%strDomain%
     ```

     Linux:

     ```
     #HOSTNAME=`hostname -f`
     ```

     Explicitly specify your computer host name.

- Security level.

  Windows:

  ```
  REM set SECURITY_LEVEL=
  ```

  Linux:

  ```
  #SECURITY_LEVEL=""
  ```

  Explicitly specify a security level to avoid future errors when starting the job scheduler.

  For security levels 2 and higher, you have to provide a password in a graphical window at the time of job submission.

Start mjs Service and One Worker

In a command-line terminal, `cd` to *`matlabroot`*`\toolbox\parallel\bin`, where *`matlabroot`* is the MATLAB Parallel Server installation folder, for instance, `C:\Program Files\MATLAB\R2023a`. Run these commands (directly at the command line or by using scripts):

```
mjs install
mjs start
startjobmanager -name JobScheduler -remotehost hostname -v
startworker -jobmanagerhost hostname -jobmanager JobScheduler
    -remotehost hostname -v
```

Here, *`hostname`* is the host name of your computer. This name is the host name that you specified in the file `mjs_def.bat` (Windows) or `mjs_def.sh` (Linux).

For more details and configuring services with multiple workers, see:

- “Install and Configure MATLAB Parallel Server for MATLAB Job Scheduler and Network License Manager” (MATLAB Parallel Server)
- `mjs`

Offload Analysis from Client Node

Once you have set up the computer that acts as the head node and worker node, you are ready to offload a Polyspace analysis from the client node (the CI server running scripts on Jenkins on another CI system).

To offload an analysis, enter:

```
polyspaceserverroot\polyspace\bin\polyspace-code-prover-server
  -batch -scheduler hostname|MJSName@hostname [options] [-mjs-username name]
```

where:

- *`polyspaceserverroot`* is the installation folder of Polyspace Server products on the client node, for instance, `C:\Program Files\Polyspace Server\R2023a`.
- *`hostname`* is the host name of the computer that hosts the head node of the MATLAB Parallel Server cluster.

  `MJSName` is the name of the MATLAB Job Scheduler on the head node host.

  If you use the `startjobmanager` command to start the MATLAB Job Scheduler, *`MJSName`* is the argument of the option `-name`.
- *`options`* are the Polyspace analysis options. These options are the same as that of a local analysis. For instance, you can use these options:

  - `-sources-list-file`: Specify a text file that has one source file name per line.
  - `-options-file`: Specify a text file that has one option per line.
  - `-results-dir`: Specify a download folder for storing results after analysis.

  For the full list of options, see "Complete List of Polyspace Code Prover Analysis Options".
- *`name`* is the user name required for job submissions using MATLAB Parallel Server. This credential is required only if you use a security level of 1 or higher for MATLAB Parallel Server submissions. See "Set MATLAB Job Scheduler Cluster Security" (MATLAB Parallel Server).

For security levels 2 and higher, you have to provide a password in a graphical window at the time of job submission. To avoid this prompt in the future, you can specify that the password be remembered on the computer.

The analysis executes locally on the CI server up to the end of the compilation phase. After compilation, the analysis job is submitted to the other server. On completion, the analysis results are downloaded back to the CI server. You can then upload the results to Polyspace Access for review, or report the results in some other format.

See Also

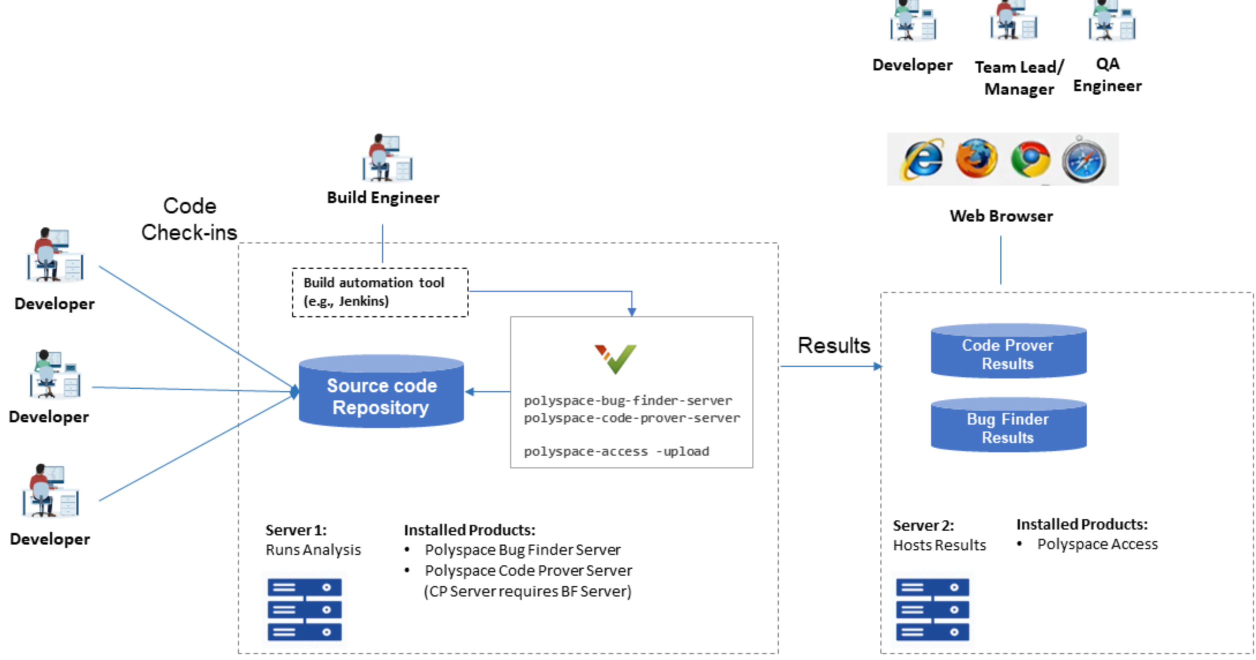
`polyspace-access`

More About

- "Run Polyspace Code Prover on Server and Upload Results to Web Interface"

Sample Scripts for Polyspace Analysis with Jenkins

In a continuous integration process, developers submit code to a shared repository. An automated build system using a tool such as Jenkins builds and tests each submission at regular intervals or based on predefined triggers and integrates the code. You can run a Polyspace analysis as part of this process.



This topic provides sample Shell scripts that run a Polyspace analysis using Polyspace Bug Finder Server and upload the results for review in the Polyspace Access web interface. The script also sends e-mail notifications to potential reviewers. Notified reviewers can login to the Polyspace Access web interface (if they have a Polyspace Access license) and review the results.

Extending Sample Scripts to Your Development Process

The scripts are written for a specific development toolchain but can be easily extended to the processes used in your project, team or organization. The scripts are also meant to be run in a Jenkins freestyle project. If you are using Jenkins Pipelines, see "Sample Jenkins Pipeline Scripts for Polyspace Analysis".

In particular, the scripts:

- *Run on Linux only.*

  The scripts use some Linux-specific commands such as `export`. However, these commands are not an integral part of the Polyspace workflow. If you write Windows scripts (`.bat` files), use the equivalent Windows commands instead.

- *Work only with Jenkins after you install the Polyspace plugin.*

  The scripts are designed for the Jenkins plugin in these two ways:

  - The scripts uses helper functions `$ps_helper` and `$ps_helper_access` for simpler scripting. The helper functions export Polyspace results for e-mail attachments and use command-line utilities to filter the results.

    These helper functions are available only with the Jenkins plugin. However, the underlying commands come with a Polyspace Bug Finder Server installation. On build automation tools other than Jenkins, you can create these helper functions using the `polyspace-report-generator` command or `polyspace-access` command (with the `-export` option). See “Send Email Notifications with Polyspace Code Prover Server Results”.

    If you perform a distributed build in Jenkins, the plugin must be installed in the same folder in the same operating system on both the master node and the agent node executing the Polyspace analysis. Otherwise, you cannot use the helper functions.
  - The scripts create text files for e-mail attachments and mail subjects and bodies for personalized e-mails. If you install the Polyspace plugin in Jenkins, an extension of an e-mail plugin is available for use in your Jenkins projects. The e-mail plugin allows you to easily send the personalized e-mails with the previously created subjects, bodies and attachments. Without the Polyspace plugin, you have to find an alternative way to send the e-mails.
- *Run a Bug Finder analysis.*

  The scripts run Bug Finder on the demo example `Bug_Finder_Example`. If you install the product Polyspace Bug Finder Server, the folder containing the demo example is *`polyspaceserverroot`*`/polyspace/examples/cxx/Bug_Finder_Example`. Here, *`polyspaceserverroot`* is the installation folder for Polyspace Server products, for instance, `/usr/local/Polyspace Server/R2019a/`.

  You can easily adapt the script to run Code Prover. Replace `polyspace-bug-finder-server` with `polyspace-code-prover-server`. You can use the demo example `Code_Prover_Example` specifically meant for Code Prover.

Prerequisites

To run a Polyspace analysis on a server and review the results in the Polyspace Access web interface, you must perform a one-time setup.

- To run the analysis, you must install one instance of the Polyspace Server product.
- To upload results, you must set up the components required to host the web interface of Polyspace Access.
- To view the uploaded results, you (and each developer reviewing the results) must have one Polyspace license.

Similar requirements apply to a Polyspace Code Prover analysis on a server.

See “Install Polyspace Server and Access Products”.

To install the Polyspace plugin, in the Jenkins interface, select **Manage Jenkins** on the left. Select **Manage Plugin**. Search for the Polyspace plugin and then download and install the plugin.

Set Up Polyspace Plugin in Jenkins

The following steps outline how to set up a Polyspace analysis in Jenkins after installing the Polyspace plugin. Note that the steps refer to Jenkins version 2.150.1. The steps in your Jenkins version and your Polyspace plugin installation might be slightly different.

If you use a different build automation tool, you can perform similar setup steps.

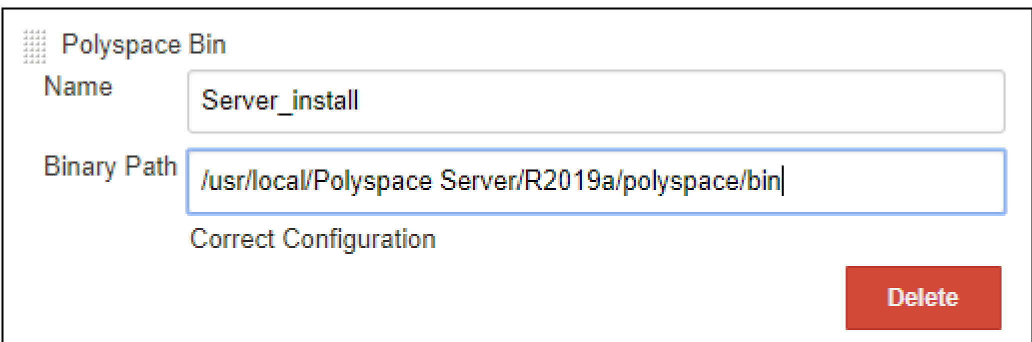
Specify Paths to Polyspace Commands and Server Details for Polyspace Access Web Interface

Specify the full paths of the folder containing the Polyspace commands and host name and port number of the server hosting the Polyspace Access web interface. After you specify the paths, in your scripts, you do not have to use the full paths to the commands or the server details for uploading results.

1. In the Jenkins interface, select **Manage Jenkins** on the left. Select **Configure System**.
2. In the **Polyspace** section, specify the following:

   - Paths to Polyspace commands.

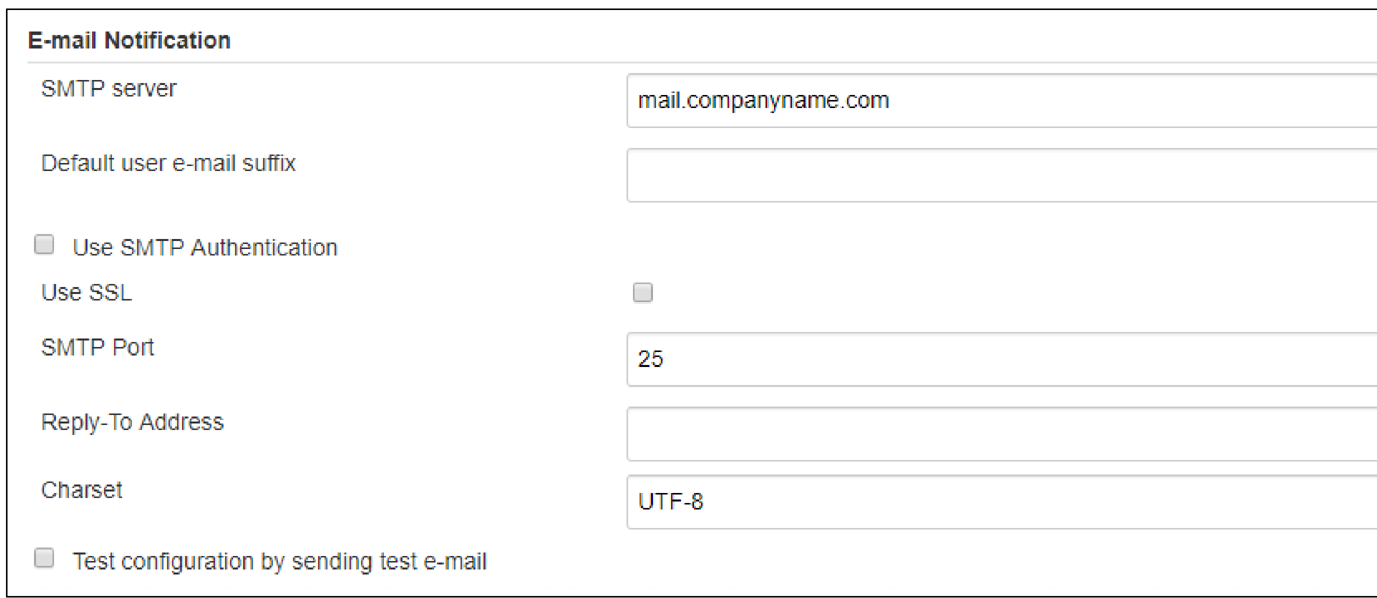
     The path refers to `*polyspaceserverroot*/polyspace/bin`, where `*polyspaceserverroot*` is the installation folder for Polyspace Server products, for instance, `/usr/local/Polyspace Server/R2019a/`.

   - The host name, port number and protocol (`http` or `https`) used by the server hosting the Polyspace Access web interface.

The **Name** field allows you to define a convenient shorthand that you use later in Jenkins projects.

3. In the **E-mail Notification** section, specify your company's SMTP server (and other details needed for sending e-mails).



Create Jenkins Project for Running Polyspace

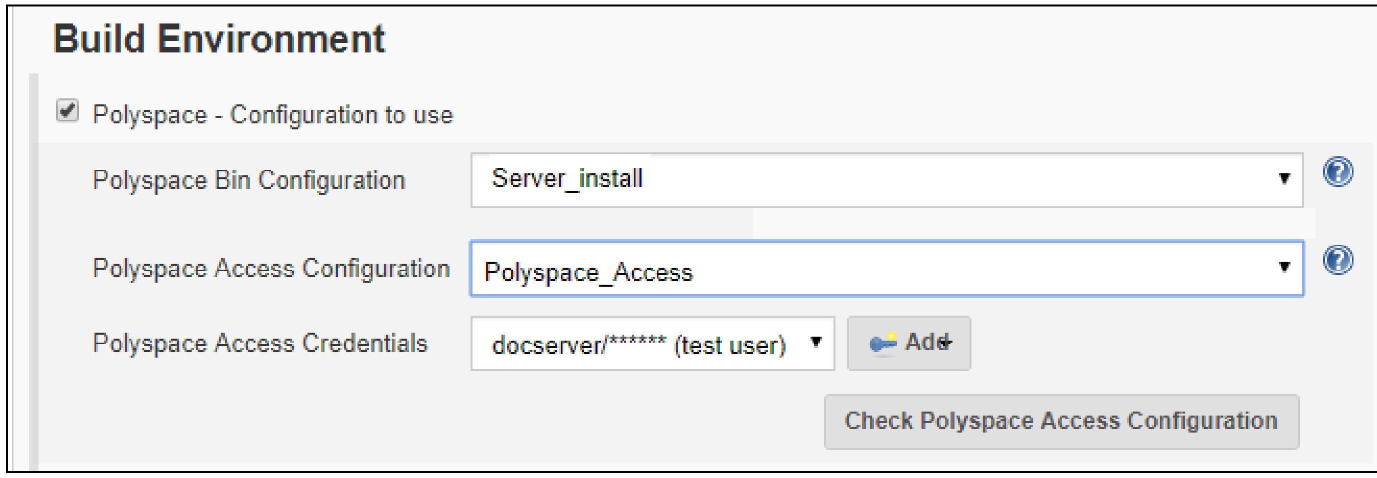
When you create a Jenkins project (for instance, a Freestyle project), you can refer to the Polyspace paths by the global shorthands that you defined earlier.

To create a Jenkins project for running Polyspace:

1. In the Jenkins interface, select **New Item** on the left. Select **Freestyle Project**.
2. In the **Build Environment** section of the project, enter the two shorthand names you defined earlier:

   - The name for the path to the folder containing the Polyspace commands
   - The name for the details of the server hosting the Polyspace Access web interface.

   Also, enter a login and password that can be used to upload to the Polyspace Access web interface. The login and password must be associated with a Polyspace Access license.



3 In the **Build** section of the project, you can enter scripts that use the Polyspace commands and details of the server hosting the Polyspace Access web interface. The scripts run a Polyspace analysis and upload results to the Polyspace Access web interface.

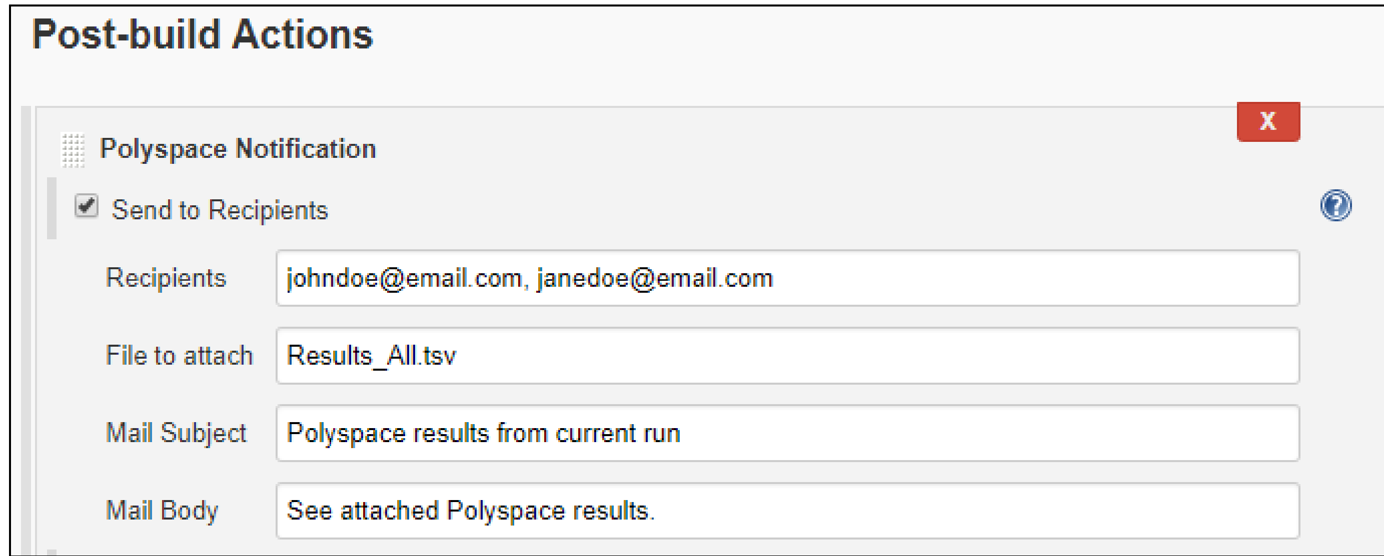
Build

Execute shell

Command

```
set -e
export RESULT=ResultBF
export PROG=Bug_Finder_Example_2
export PARENT_PROJECT=testProject
rm -rf Notification && mkdir -p Notification


build_cmd="gcc -c sources/*.c"
polyspace-configure \
      -allow-overwrite \
      -allow-build-error \
      -prog $PROG \
      -author jenkins \
      -output-options-file $PROG.psopts \
      $build_cmd

polyspace-bug-finder-server -options-file $PROG.psopts -results-dir $RESULT
```

4 In the **Post-build Actions** section of the project, configure e-mail addresses and attachments to be sent after the analysis.



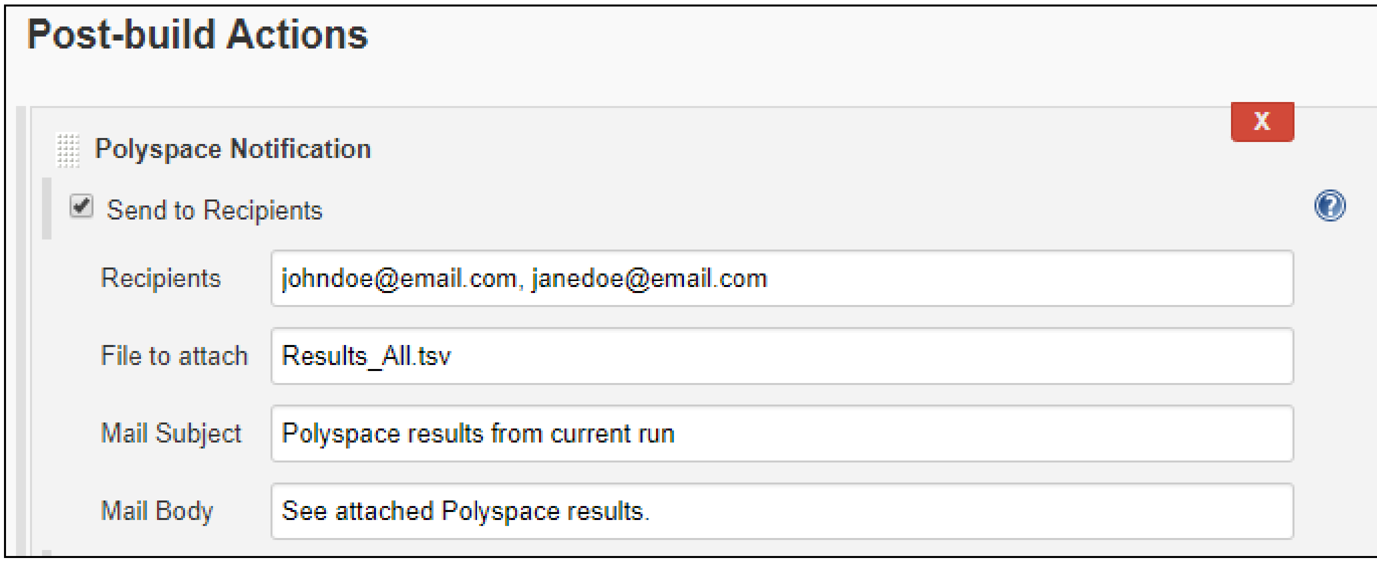
Script to Run Bug Finder, Upload Results and Send Common Notification

This script runs a Bug Finder analysis, uploads the results and exports defects with high impact for a common notification email to all recipients.

The script assumes that the current folder contains a folder `sources` with `.c` files. Otherwise modify the line `gcc -c sources/*.c` with the full path to the sources.

```
set -e
export RESULT=ResultBF
export PROG=Bug_Finder_Example
export PARENT_PROJECT=/public/BugFinderExample_PRS_01

# ================================================================
# Trace build command and create an options file

build_cmd="gcc -c sources/*.c"
polyspace-configure \
      -allow-overwrite \
      -allow-build-error \
      -prog $PROG \
      -author jenkins \
      -output-options-file $PROG.psopts \
      $build_cmd


# ================================================================
# Run Bug Finder on the options file

polyspace-bug-finder-server -options-file $PROG.psopts -results-dir $RESULT

# ================================================================
# Upload results to Polyspace Access web interface

$ps_helper_access -create-project $PARENT_PROJECT
$ps_helper_access \
      -upload $RESULT \
      -parent-project $PARENT_PROJECT \
      -project $PROG

# ================================================================
# Export results filtered for defects with "High" impact

$ps_helper_access \
      -export $PARENT_PROJECT/$PROG \
      -output Results_All.tsv \
      -defects High

# ================================================================
# Finalize Jenkins status

exit 0
```

After the script is run, you can create a post-build action to send an e-mail to all recipients with the exported file `Results_All.tsv`.



In this script, `$ps_helper_access` is a shorthand for the `polyspace-access` command with the options specifying host name, port, login and encrypted password included. The other `polyspace-access` options are explicitly written in the script.

Script to Run Bug Finder, Upload Results and Send Personalized Notification

This script runs the previous Bug Finder analysis and uploads the results. However, the script differs from the previous script in these ways:

- The script uses a `run_command` function that prints a message when running a command. The function helps determine from the console output which part of the script is running.
- When exporting the results, the script creates a separate results file for different owners.
  - A main file `Results_All.tsv` contains all results. This file is sent in e-mail attachment to a manager. The manager email is configured in the post-build step.

    If the file contains more than 10 defects, the build status is considered as a failure. The script sends a status `UNSTABLE` in the e-mail notification.
  - The results file `Results_Users_userA.tsv` exported for `userA` contains defects from the group Programming and with impact High.

    This result file is sent in e-mail attachment to `userA`.
  - The results file `Results_Users_userB.tsv` exported for `userB` contains defects from the function `bug_memstdlib()`.

    This result file is sent in e-mail attachment to `userB`.
- A separate mail subject is created for the manager in the file `mailsubject_manager.txt` and for users `userA` and `userB` in the files `mailsubject_user_userA.txt` and `mailsubject_user_userB.txt` respectively.

  A mail body is created for the email to the manager in the file `mailbody_manager.txt`.

The script:

- Assumes that the current folder contains a folder `sources` with `.c` files.

  Otherwise, modify the line `gcc -c sources/*.c` with the full path to the sources.
- Assumes users named `userA` and `userB`. In particular, the email addresses `userA@companyname.com` and `userB@companyname.com` (determined from the user name and SMTP server configured earlier) must be real e-mail addresses.

  Replace the names with real user names.

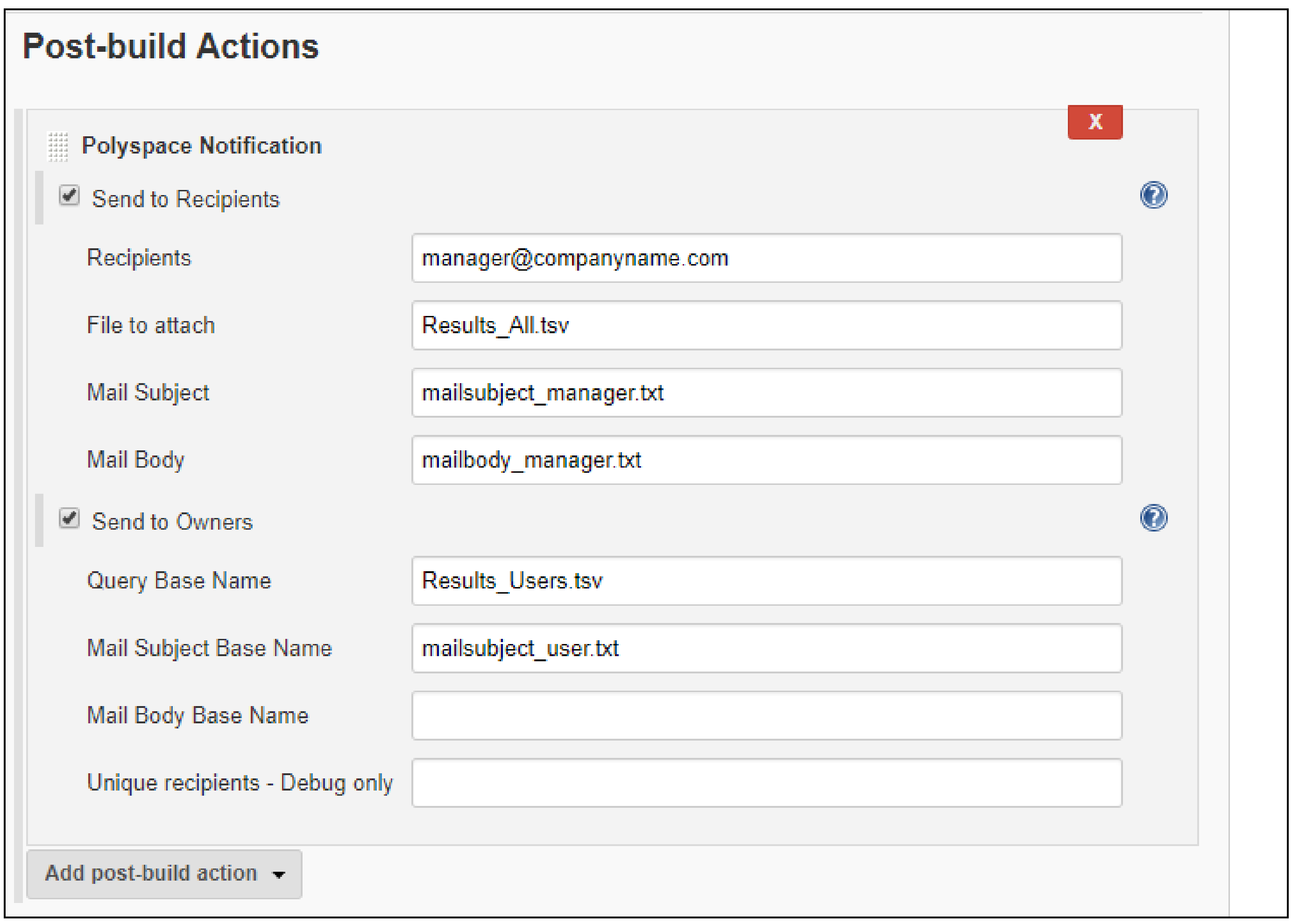
```
set -e
export RESULT=ResultBF
export PROG=Bug_Finder_Example
export REPORT=Results_List.tsv


# ================================================================
# Define function to print message while running command
run_command()
{
# $1 is a message
# $2 $3 ... is the command to dump and to run
message=$1
shift
cat >> mailbody_manager.txt << EOF
$(date): $message

EOF
"$@"
}

# ================================================================
# Initialize mail body
cat > mailbody_manager.txt << EOF
Dear Manager(s)

Here is the report of the Jenkins Job ${JOB_NAME} #${BUILD_NUMBER}
It contains all Red Defect found in Bug Finder Example project

EOF

# ================================================================
# Trace build command and create options file

build_cmd="gcc -c sources/*.c"
run_command "Tracing build command",               \
            polyspace-configure                   \
                -allow-overwrite                  \
                -allow-build-error                \
                -prog $PROG                       \
                -author jenkins                   \
                -output-options-file $PROG.psopts \
                    $build_cmd

# ================================================================
# Run Bug Finder on the options file

run_command "Running Bug finder" \
            polyspace-bug-finder-server -options-file $PROG.psopts\
            -results-dir $RESULT



# ================================================================
# Upload results to Polyspace Access web interface

run_command "Creating Project $PARENT_PROJECT" \
```

```
  $ps_helper_access -create-project $PARENT_PROJECT

run_command "Uploading on $PARENT_PROJECT/$PROG" \
  $ps_helper_access \
        -upload $RESULT \
        -parent-project $PARENT_PROJECT \
        -project $PROG \
        -output upload.output
PROJECT_RUNID=$($ps_helper prs_print_runid upload.output)
PROJECT_URL=$($ps_helper prs_print_projecturl upload.output $POLYSPACE_ACCESS_URL)

# ==================================================================
# Export report

run_command "Exporting report from $PARENT_PROJECT/$PROG" \
  $ps_helper_access \
        -export $PROJECT_RUNID \
        -output $REPORT \
        -defects High


# ==================================================================
# Filter Reports

run_command "Filtering reports for defects" \
            $ps_helper report_filter \
                $REPORT \
                Results_All.tsv \
                Family Defect \


# ==================================================================
# Filter Reports for userA and userB

run_command "Filtering Reports for userA based on Group and Information" \
            $ps_helper report_filter \
                $REPORT \
                Results_Users.tsv \
                userA \
                Group Programming \
                Information "Impact: High"
run_command "Filtering Reports for userB based on Function" \
            $ps_helper report_filter \
                $REPORT \
                Results_Users.tsv \
                userB \
                Function "bug_memstdlib()"


# ==================================================================
# Update Jenkins status
# Jenkins build status is unstable when there are more than 10 Defects

BUILD_STATUS=$($ps_helper report_status Results_All.tsv 10)

# ==================================================================
# Update mail body and mail subject
```

```
NB_FINDINGS_ALL=$($ps_helper report_count_findings Results_All.tsv)
NB_FINDINGS_USERA=$($ps_helper report_count_findings Results_Users_userA.tsv)
NB_FINDINGS_USERB=$($ps_helper report_count_findings Results_Users_userB.tsv)
cat >> mailbody_manager.txt << EOF

Number of defects: $NB_FINDINGS_ALL
Number of findings owned by userA: $NB_FINDINGS_USERA
Number of findings owned by userB: $NB_FINDINGS_USERB

All results are uploaded in: $PROJECT_URL

Build Status: $BUILD_STATUS

EOF

cat >> mailsubject_manager.txt << EOF
Polyspace run completed with status $BUILD_STATUS and $NB_FINDINGS_ALL findings
EOF

for user in userA userB
do
echo "$user - $($ps_helper report_count_findings Results_Users_$user.tsv)) findings"\
      > mailsubject_user_$user.txt
done



# ================================================================
# Exit with correct build status

[ "$BUILD_STATUS" != "SUCCESS" ] && exit 129
exit 0
```

After the script is run, you can create a post-build action to send an e-mail to a manager with the exported file `Results_All.tsv`. Specify the e-mail address in the **Recipients** field, the email subject in the **Mail Subject** field and the email body in the **Mail Body** field.

In addition, a separate e-mail is sent to `userA` and `userB` with the files `Results_Users_userA.tsv` and `Results_Users_userB.tsv` in attachment (and the content of `mailsubject_user_userA.txt` and `mailsubject_user_userB.txt` as mail subjects). The e-mail addresses are `userA@companyname.com` and `userB@companyname.com` (determined from the user name and SMTP server configured earlier).



The script uses the helper function `$ps_helper` to filter the results based on group, impact and function. The helper function uses command-line utilities to filter the main file for results and perform actions such as create a separate results file for each owner. The function takes these actions as arguments:

- `report_filter`: Filters results from exported text file based on contents of the text file.

  For instance:

  ```
  $ps_helper report_filter \
                  Results_List.tsv \
                  Results_Users.tsv \
                  userA \
                  Group Programming \
                  Information "Impact: High"
  ```

  reads the file `Results_List.tsv` and writes to the file `Results_Users_userA.tsv`. The text file `Results_List.tsv` contains columns for `Group` and `Information`. Only those rows where the `Group` column contains `Programming` and the `Information` column contains `Impact: High` are written to the file `Results_Users_userA.tsv`.
- `report_status`: Returns `UNSTABLE` or `SUCCESS` based on the number of results in a file.

For instance:

```
BUILD_STATUS=$($ps_helper report_status Results_All.tsv 10))
```

returns `UNSTABLE` if the file `Results_All.tsv` contains more than 10 results (10 rows).

- `report_count_findings`: Reports number of results in a file.

  For instance:

  ```
  NB_FINDINGS_ALL=$($ps_helper report_count_findings Results_All.tsv)
  ```

  returns the number of results (rows) in the file `Results_All.tsv`.
- `prs_print_projecturl`: Uses the host name and port number to create the URL of the Polyspace Access web interface.

  For instance:

  ```
  PROJECT_URL=$($ps_helper prs_print_projecturl Results_All.tsv $POLYSPACE_ACCESS_URL)
  ```

  reads the file `Results_All.tsv` (exported by the `polyspace-access` command) and extracts the URL of the Polyspace Access web interface in `$POLYSPACE_ACCESS_URL` and the URL of the current project in `$PROJECT_URL`.

See Also

`polyspace-bug-finder-server` | `polyspace-code-prover-server` | `polyspace-report-generator` | `polyspace-access` | `polyspace-configure`

More About

- “Run Polyspace Code Prover on Server and Upload Results to Web Interface”
- “Send Email Notifications with Polyspace Code Prover Server Results”
- “Sample Jenkins Pipeline Scripts for Polyspace Analysis”
- “Offload Polyspace Analysis from Continuous Integration Server to Another Server” on page 10-12

Sample Jenkins Pipeline Scripts for Polyspace Analysis

Jenkins Pipelines enable automating the workflow of a continuous delivery pipeline through scripts in Jenkins. You can write Pipeline scripts that build projects, run test suites and perform all necessary checks before your code is ready for shipping. You can check in these scripts as part of a version control system and subject them to the same review and versioning as the code itself.

You can run a Polyspace analysis in a Jenkins Pipeline script. If you are using Freestyle Projects instead of Pipelines in Jenkins, use the Polyspace plugin for scripting conveniences. See “Sample Scripts for Polyspace Analysis with Jenkins”. If you are using Pipelines, modify the script provided below to run a Polyspace analysis.

Prerequisites

To run a Polyspace analysis on a server and review the results in the Polyspace Access web interface, you must perform a one-time setup.

- To run the analysis, you must install one instance of the Polyspace Server product.
- To upload results, you must set up the components required to host the web interface of Polyspace Access.
- To view the uploaded results, you and each developer reviewing the results must have one Polyspace license.

See “Install Polyspace Server and Access Products”.

Run Polyspace Analysis in Stages in a Pipeline Script

To create a Jenkins Pipeline script:

1. In the Jenkins interface, select **New Item** on the left. Select **Pipeline**.
2. In the **Pipeline** section of the project, select `Pipeline script` for **Definition**. Enter this script.

   The parts in bold indicate places where you have to modify the script for your source code and Polyspace installation.

   The script is not available in the PDF documentation. Search for `Polyspace Jenkins Pipelines` in the MathWorks online documentation and copy the script from the online version of this page.

When you build this project, you can see the various stages of the analysis like this:

Prepare	Checkout	Configure	Analyze	Upload	Notification
1s	1s	14s	4min 22s	1min 32s	369ms
1s	1s	14s	4min 22s	1min 32s	369ms

This script can be part of a larger script that you save in a Jenkinsfile and commit to your version control system. See Using a Jenkinsfile.

You can modify the script as needed:

- The script runs each step of the Polyspace analysis workflow in a separate `stage` section. You can combine several steps together in one `stage`.
- The script runs Linux Shell commands by using the `sh` directive. You can run Windows commands by using the `bat` directive instead.
- The script uses data from the Credentials plugin to extract user name and password. If you save credentials in some other form, you can replace the `withCredentials` command that binds user credentials to variables.
- The script builds source code using a makefile on a Git sandbox with this `make` command:

  ```
  make -C $git_sandbox
  ```

  If you use a different build command, you can replace this line with your build command.

For more information on the Pipeline-specific syntax in this script, see:

- Pipeline Syntax: Describes `node`, `stage`, `label`.
- Pipeline Steps Reference: Describes `sh`, `mail`.
- Credentials Binding Plugin: Describes `withCredentials`.

For more information on the Polyspace commands in this script, see:

- `polyspace-configure`
- `polyspace-bug-finder-server` (also `polyspace-code-prover-server`)
- `polyspace-access`

See Also

"Sample Scripts for Polyspace Analysis with Jenkins"

Integrate Polyspace Server Products with MATLAB

You can install Polyspace Bug Finder Server and Polyspace Code Prover Server as standalone products and analyze C/C++ code.

When installing Polyspace server products and MATLAB, you cannot install MATLAB and Polyspace server products together in a single run of the installer. First install MATLAB by running the MATLAB installer. Then install the Polyspace server product in a different root folder by running the installer separately. For instance, in Windows:

- Your default MATLAB root folder is `C:\Program Files\MATLAB\R2023a`.
- Your default Polyspace root folder is `C:\Program Files\Polyspace Server\R2023a` for the Polyspace server products.

To automate the Polyspace analysis by using MATLAB scripts, integrate the Polyspace server products and MATLAB by running a post-installation step.

Integrate Polyspace Server Products with MATLAB

You can integrate your Polyspace server product with MATLAB only if both installations are from the same release. After the integration, you can use all MATLAB functions and classes available for running Polyspace.

To link your MATLAB and Polyspace installations:

1. Open MATLAB with administrator privileges.
2. At the MATLAB command prompt, enter:

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

   By default, Polyspace is installed in the folder `C:\Program Files\Polyspace\R2023a`. If you install Polyspace in the default folder, the command integrates Polyspace with MATLAB. If a Polyspace installation is not detected at the default location, provide the path to the Polyspace installation folder when prompted. The process might take a few minutes to complete.

   To avoid the prompt during installation, enter:

   ```
   polyspacesetup('install', 'polyspaceFolder', Folder, 'silent', true);
   ```
3. Restart MATLAB. You can now use all functions and classes available for running Polyspace server products.

A MATLAB installation can be integrated with only one Polyspace installation. To integrate to a new Polyspace installation, any previous integration must be removed. To remove the integration between a Polyspace and MATLAB installation, open MATLAB with administrator privilege and at the MATLAB command prompt, enter:

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

Check Integration Between MATLAB and Polyspace

To check if a MATLAB installation is already integrated with a Polyspace installation, open MATLAB and at the command prompt, enter:

```
ver
```

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

The MATLAB-Polyspace integration adds some Polyspace installation subfolders to the MATLAB search path. To see which paths were added, enter:

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

Run Polyspace Server Products with MATLAB Scripts

In a continuous integration process, you can execute MATLAB scripts that run a Polyspace analysis on new code submissions and compares the results against predefined criteria. Use these functions/classes:

- Create a `polyspace.Project` object to configure Polyspace analysis options, run an analysis and read results to MATLAB tables. You can use other MATLAB functions for comparing results against predefined criteria.

  To only read existing results without running an analysis, use the `polyspace.CodeProverResults` class with the path to a results folder.
- If you want a more granular selection of checkers for:
  - Coding rules, create a `polyspace.CodingRulesOptions` object.
  - Bug Finder defects, create a `polyspace.DefectsOptions` object.

  To create a custom target for the analysis and explicitly specify sizes of data types, create a `polyspace.GenericTargetOptions` object.

You can also use the `polyspaceCodeProverServer` function to run the analysis and then read results with the `polyspace.CodeProverResults` class. If you use build commands to build your source code, you can create a Polyspace configuration from the build command using the `polyspaceConfigure` function.

See Also

`polyspacesetup`

Configure Job Submissions from Desktop to Server

Offload Polyspace Analysis to Remote Servers from Desktop

- “Send Polyspace Analysis from Desktop to Remote Servers” on page 11-2
- “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts” on page 11-5

Send Polyspace Analysis from Desktop to Remote Servers

In this section...

"Client-Server Workflow for Running Analysis" on page 11-2
"Prerequisites" on page 11-3
"Offload Analysis in Polyspace User Interface" on page 11-3

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 11-5.
- For a simple tutorial that walks through all the steps for offloading a Polyspace analysis, see "Send Code Prover Analysis from Desktop to Locally Hosted Server". In the tutorial, the same computer acts as the client and the server.

Client-Server Workflow for Running Analysis

After the initial setup, you can submit a Polyspace analysis from a client desktop to a server. The client-server workflow happens in three steps. All three steps can be performed on the same computer or three different computers.

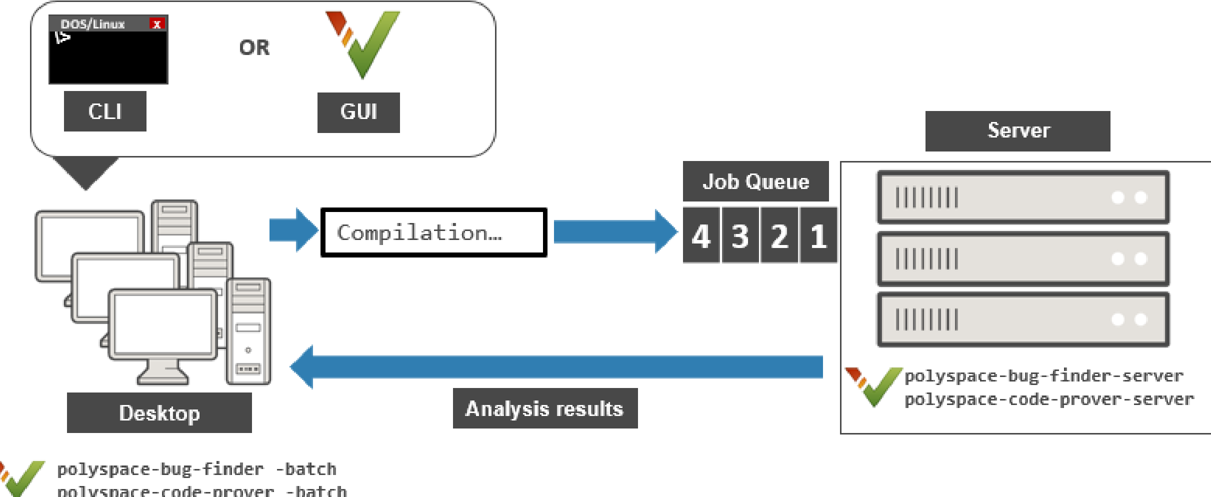
1. **Client node**: You specify Polyspace analysis options and start the analysis on the client desktop. The initial phase of analysis up to compilation runs on the desktop. After compilation, the analysis job is submitted to the server.

   You require the Polyspace desktop product, Polyspace Bug Finder on the computer that acts as the client node.
2. **Head node**: The server consists of a head node and several worker nodes. The head node uses a job scheduler to manage submissions from multiple client desktops. The jobs are then distributed to the worker nodes as they become available.

   You require the product MATLAB Parallel Server on the computer that acts as the head node.
3. **Worker nodes**: When a worker becomes available, the job scheduler assigns the analysis to the worker. The Polyspace analysis runs on the worker and the results are downloaded back to the client desktop for review.

   You require the product MATLAB Parallel Server on the computers that act as worker nodes. You also require the Polyspace server products, Polyspace Bug Finder Server and/or Polyspace Code Prover Server, to run the analysis.

Note The versions of Polyspace on the client and worker nodes must match.



Prerequisites

Before offloading an analysis from the user interface of the Polyspace desktop products, you must set up your project's source files, analysis options, and remote analysis settings. If you have not done so, for more information on:

- How to add source files, see "Add Source Files for Analysis in Polyspace Desktop User Interface" on page 2-2.
- How to set up communication between client and server, see "Install Products for Submitting Polyspace Analysis from Desktops to Remote Server".

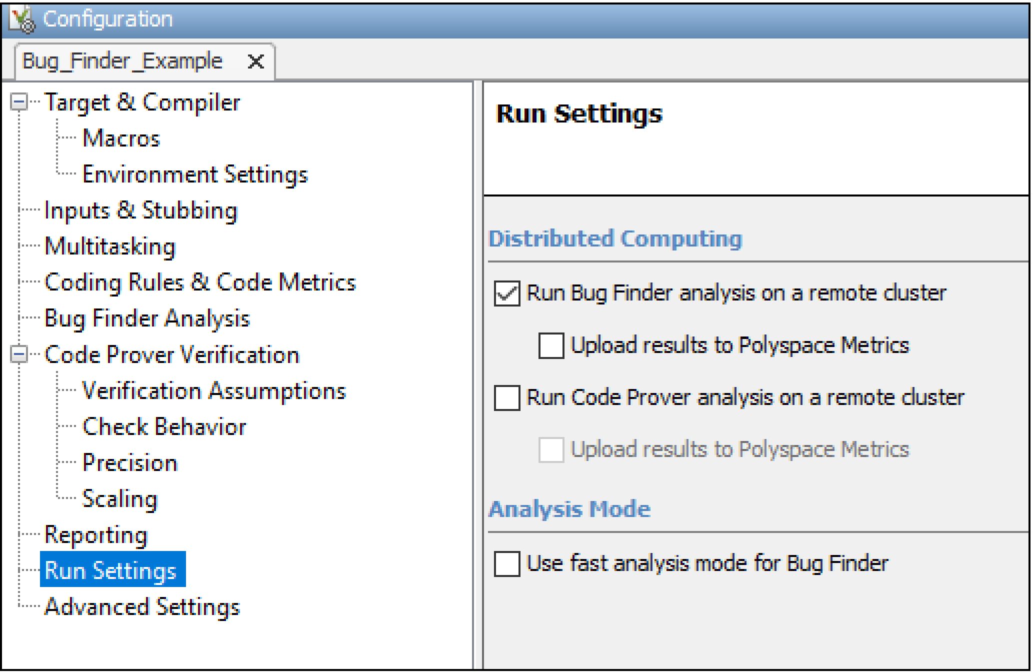
Once you have set up a Polyspace project and established communicated between a desktop and a remote server, you are ready to offload a Polyspace analysis.

Offload Analysis in Polyspace User Interface

To start a remote analysis:

1. Select a project to analyze.
2. On the **Configuration** pane, select **Run Settings**.

   Select **Run Bug Finder analysis on a remote cluster** and/or **Run Code Prover analysis on a remote cluster**.



3. Start the analysis. For instance, to start a Bug Finder analysis, click the **Run Bug Finder** button.

   The compilation part of the analysis takes place on the desktop product. After compilation, the analysis is offloaded to the server.
4. To monitor the analysis, select **Tools > Open Job Monitor**. In the Polyspace Job Monitor, follow your queued job to monitor progress.

   Once the analysis is complete, the results are downloaded back to the user interface of the Polyspace desktop products. You can open the results directly in the user interface.

   If the analysis stops after compilation and you have to restart the analysis, to avoid restarting from the compilation phase, use the option `-submit-job-from-previous-compilation-results`.

See Also

`Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`

More About

- “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”
- “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts” on page 11-5

Send Polyspace Analysis from Desktop to Remote Servers Using Scripts

Instead of running a Polyspace analysis on your local desktop, you can send the analysis to a remote cluster. You can use a dedicated cluster for running Polyspace to free up memory on your local desktop.

This topic shows how to use Windows or Linux scripts to send the analysis to a remote cluster and download the results to your desktop after analysis.

- To offload an analysis from the Polyspace user interface, see “Send Polyspace Analysis from Desktop to Remote Servers” on page 11-2.
- For a simple tutorial that walks through all the steps for offloading a Polyspace analysis, see “Send Code Prover Analysis from Desktop to Locally Hosted Server”. In the tutorial, the same computer acts as the client and the server.

Client-Server Workflow for Running Analysis

After the initial setup, you can submit a Polyspace analysis from a client desktop to a server. The client-server workflow happens in three steps. All three steps can be performed on the same computer or three different computers.

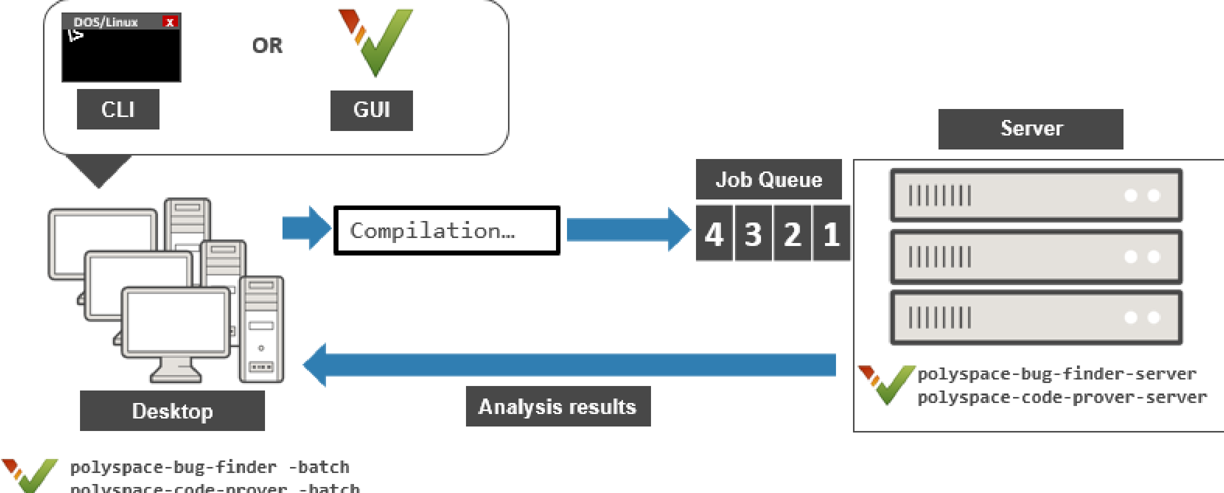
1. **Client node**: You specify Polyspace analysis options and start the analysis on the client desktop. The initial phase of analysis up to compilation runs on the desktop. After compilation, the analysis job is submitted to the server.

   You require the Polyspace desktop product, Polyspace Bug Finder on the computer that acts as the client node.
2. **Head node**: The server consists of a head node and several worker nodes. The head node uses a job scheduler to manage submissions from multiple client desktops. The jobs are then distributed to the worker nodes as they become available.

   You require the product MATLAB Parallel Server on the computer that acts as the head node.
3. **Worker nodes**: When a worker becomes available, the job scheduler assigns the analysis to the worker. The Polyspace analysis runs on the worker and the results are downloaded back to the client desktop for review.

   You require the product MATLAB Parallel Server on the computers that act as worker nodes. You also require the Polyspace server products, Polyspace Bug Finder Server and/or Polyspace Code Prover Server to run the analysis.

Note The versions of Polyspace on the client and worker nodes must match.



Prerequisites

Before you run a remote analysis by using scripts, you must set up communication between a desktop and a remote server. See "Install Products for Submitting Polyspace Analysis from Desktops to Remote Server".

Run Remote Analysis

To run a remote analysis, use the following command. Here, `[]` indicates optional flags.

```
polyspaceroot\polyspace\bin\polyspace-code-prover
  -batch -scheduler NodeHost|MJSName@NodeHost [-wait -download]
  [options] [-mjs-username name]
```

where:

- *`polyspaceroot`* is the installation folder of Polyspace desktop products, for instance, `C:\Program Files\Polyspace\R2023a`.
- *`NodeHost`* is the name of the computer that hosts the head node of the MATLAB Parallel Server cluster.

  `MJSName` is the name of the MATLAB Job Scheduler on the head node host.

  If you set up communications with a cluster from the Polyspace user interface, you can determine *`NodeHost`* and *`MJSName`* from the user interface.

  Select **Tools** > **Preferences**, and then click **Settings** on the **Server Configuration** tab to open the **Cluster Profile Manager**. Select the cluster profile in the left pane, and see the **MJSName** and **Host** fields on the **Properties** tab for *`MJSName`* and *`NodeHost`*.

If you use the `startjobmanager` command to start the MATLAB Job Scheduler, *`MJSName`* is the argument of the option `-name`. For details, see “Configure Advanced Options for MATLAB Job Scheduler Integration” (MATLAB Parallel Server).

- *`options`* are the analysis options. These options are the same as that of a local analysis. For instance, you can use these options:

  - `-sources-list-file`: Specify a text file with one source file name per line.
  - `-options-file`: Specify a text file with one option per line.
  - `-results-dir`: Specify a download folder for storing results after analysis.

  For the full list of options, see “Complete List of Polyspace Code Prover Analysis Options”. Alternatively, you can:

  - Start an analysis in the user interface and stop after compilation. You can obtain the text files and scripts for running the analysis at the command line. See “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 4-17.
  - Enter `polyspace-codeprover -h`. The list of available options with a brief description are displayed.
  - Place your cursor over each option on the **Configuration** pane in the Polyspace user interface. Click the **More Help** button for information on the option syntax and when the option is required.
- *`name`* is the username required for job submissions using MATLAB Parallel Server. These credentials are required only if you use a security level of 1 or higher for MATLAB Parallel Server submissions. See “Set MATLAB Job Scheduler Cluster Security” (MATLAB Parallel Server).

  For security levels 2 and higher, you have to provide a password in a graphical window at the time of job submission. To avoid this prompt in the future, you can specify that the password be remembered on the computer.

The analysis happens in two parts:

1. The first part of the analysis up to the end of the compilation phase executes locally on your desktop. After compilation, the software submits the analysis job to the cluster and provides a job ID. You can also read the ID from the file `ID.txt`, which is stored in the `.status` subfolder of the results folder. To monitor your analysis, use the `polyspace-jobs-manager` command with the job ID.
2. The remaining part of the analysis continues on the cluster. The command waits till the analysis is completed and the results automatically downloaded back to the desktop. If you want to free up the console and download results later using the `polyspace-jobs-manager` command, omit the options `-wait -download`.

If the analysis stops after compilation and you have to restart the analysis, to avoid rerunning the compilation phase, use the option `-submit-job-from-previous-compilation-results`.

Manage Remote Analysis

To manage multiple remote analyses, use the option `-batch`. For instance:

```
polyspaceroot\polyspace\bin\polyspace-jobs-manager action
         -scheduler schedulerName
```

See also `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`. Here:

- *`polyspaceroot`* is your MATLAB installation folder.
- *`schedulerName`* is one of the following:
  - Name of the computer that hosts the head node of your MATLAB Parallel Server cluster (*`NodeHost`*).
  - Name of the MATLAB Job Scheduler on the head node host (*`MJSName@NodeHost`*).
  - Name of a MATLAB cluster profile (*`ClusterProfile`*).

    For more information about clusters, see “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox)

  If you do not specify a job scheduler, `polyspace-job-manager` uses the scheduler specified in the Polyspace preferences. To see the scheduler name, select **Tools** > **Preferences**. On the **Server Configuration** tab, see the **Job scheduler host name**.
- *`action`* refers to the possible action commands to manage jobs on the scheduler:
  - `listjobs`:

    Generate a list of Polyspace jobs on the scheduler. For each job, the software produces this information:

    - `ID` — Verification or analysis identifier.
    - `AUTHOR` — Name of user that submitted job.
    - `APPLICATION` — Name of Polyspace product, for example, Polyspace Code Prover or Polyspace Bug Finder.
    - `LOCAL_RESULTS_DIR` — Results folder on local computer, specified through the **Tools** > **Preferences** > **Server Configuration** tab.
    - `WORKER` — Local computer from which job was submitted.
    - `STATUS` — Status of job, for example, `running` and `completed`.
    - `DATE` — Date on which job was submitted.
    - `LANG` — Language of submitted source code.
  - `download -job` *`ID`* `-results-folder` *`FolderPath`*:

    Download results of analysis with specified ID to folder specified by *`FolderPath`*. If you use the option `-wait -download` when sending the analysis job to a server, the results are automatically downloaded after analysis. Only when you want to explicitly download results do you need to use the `polyspace-jobs-manager` command with the `download` action.

    When the analysis job is queued on the server, the command `polyspace-code-prover` returns a job id. In addition, a file `ID.txt` that is stored in the `.status` subfolder of the results folder contains the job ID in this format:

    `job_id``;`*`server_name`*`:`*`project_name version_number`*

    For instance, `92;localhost:Demo 1.0`.

    If you do not use the `-results-folder` option, the software downloads the result to the folder that you specified when starting analysis, using the `-results-dir` option.

After downloading results, use the Polyspace user interface to view the results.

- `getlog -job` *`ID`*:

  Open log for job with specified ID.
- `remove -job` *`ID`*:

  Remove job with specified ID.
- `promote -job` *`ID`*:

  Promote job with specified ID in the queue.
- `demote -job` *`ID`*

  Demote job with specified ID in the queue.

Sample Scripts for Remote Analysis

In Windows, to avoid typing the commands each time, you can save the commands in a batch file. In Linux, you can relaunch the analysis by using a shell script. To create a batch file for running analysis:

1. Save your analysis options in a file `listofoptions.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\listofoptions.txt
   "%POLYSPACE_PATH%\polyspace-code-prover.exe" -batch -scheduler hostname
                             -results-dir %RESULTS_PATH% -options-file %OPTIONS_FILE%
   pause
   ```

   `polyspaceroot` is the Polyspace installation folder. *`hostname`* is the name of the computer that hosts the head node of your MATLAB Parallel Server cluster.
3. Replace the definitions of these variables in the file:

   - `POLYSPACE_PATH`: Enter the actual location of the `.exe` file.
   - `RESULTS_PATH`: Enter the path to a folder. The files generated during compilation are saved in the folder.
   - `OPTIONS_FILE`: Enter the path to the file `listofoptions.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 11-2

Configuration Workflows Common to All Platforms

Configure Polyspace Analysis

- “Specify Polyspace Analysis Options” on page 12-2
- “Options Files for Polyspace Analysis” on page 12-5

Specify Polyspace Analysis Options

You can change the default options associated with a Polyspace analysis. For instance, you can:

- Change the set of defects that Bug Finder looks for.

  See `Find defects (-checkers -disable-checkers)`.
- Change the default behavior of run-time checkers in Code Prover.

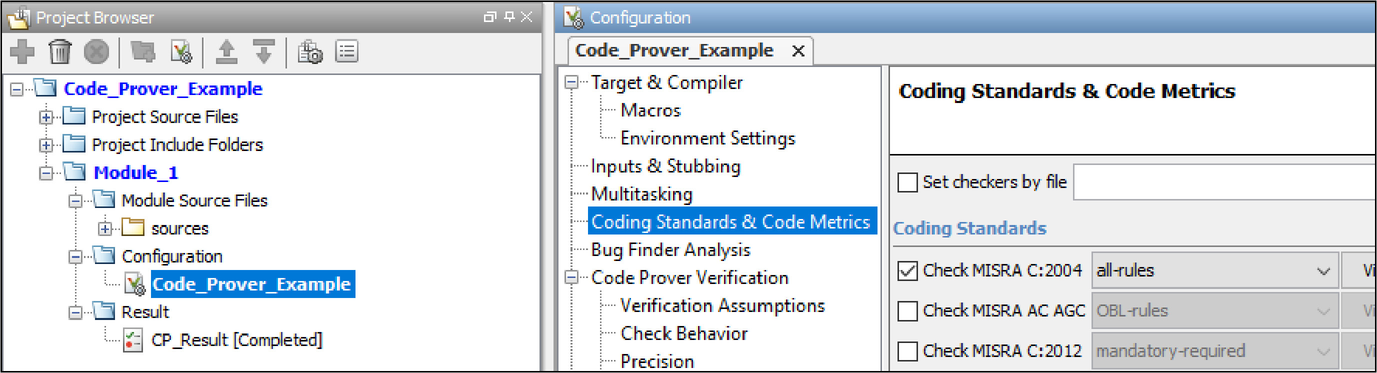
  See, for instance, `Overflow mode for unsigned integer (-unsigned-integer-overflows)`.

For the full list of analysis options, see "Complete List of Polyspace Code Prover Analysis Options".

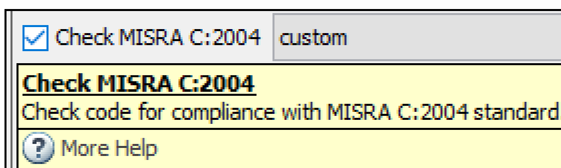
Depending on how you run Polyspace, you can configure the analysis options accordingly.

Polyspace User Interface

In the Polyspace user interface, you create a project for the analysis. The project can have one or more modules. Click the **Configuration** node in a module. On the **Configuration** pane, change options as needed.



For more information, see the tooltip on each option. Click the **More help** link for context-sensitive help on the options.



For more information, see "Run Analysis in Polyspace Desktop User Interface" on page 3-2.

Windows or Linux Scripts

Provide the options to the `polyspace-bug-finder` or `polyspace-code-prover` command. See also:

- `polyspace-bug-finder`

- `polyspace-code-prover`

For instance:

```
polyspace-code-prover -sources file_name \
        -main-generator main-generator-writes-variables all
```

You can also provide the options in a text file. See “Run Polyspace Analysis from Command Line” on page 4-2.

MATLAB Scripts

Create a `polyspace.Project` object and set the options through the `Configuration` property of the object. See also:

- `polyspace.Project`
- polyspace.Project.Configuration Properties

For instance:

```
proj = polyspace.Project;
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.BugFinderAnalysis.EnableCheckers = false;
```

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

Eclipse and Eclipse-based IDEs

Select **Polyspace** > **Configure Project**. Set the options in the Configuration window.

Some Target & Compiler options are automatically extracted from your Eclipse project. See “Run Polyspace Analysis on Eclipse Projects” on page 9-2.

Simulink

In your Simulink model, specify the basic options through Simulink Configuration Parameters. On the **Apps** tab, select **Polyspace** and then on the **Polyspace** tab, select **Settings**.

To navigate to Polyspace analysis options related to the generated code, on the **Polyspace** tab, see **Settings** > **Project Settings**.

See:

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 6-2
- “Configure Polyspace Options in Simulink” on page 6-53

MATLAB Coder App

In the MATLAB Coder app, after code generation, specify the basic options through the **Polyspace** pane. From this window, you can navigate to the full set of Polyspace analysis options.

See:

- “Run Polyspace on C/C++ Code Generated from MATLAB Code” on page 7-2
- “Configure Advanced Polyspace Options in MATLAB Coder App” on page 7-7

Options Files for Polyspace Analysis

To adapt the Polyspace analysis configuration to your development environment and requirements, you have to modify the default configuration through command-line options such as `-compiler`. Options files are a convenient way to collect multiple options together and reuse them across projects.

What are Options Files

Options files are text files with one option per line. For instance, the content of an options file can look like this:

```
# Options for Polyspace analysis
# Options apply to all projects in Controller module
-compiler visual16.x
-D _WIN32
-code-behavior-specifications "Z:\utils\polyspace\forbiddenfunctions.xml"
```

The lines starting with # represent comments for better readability. These lines are ignored during analysis.

Specifying Options Files

Depending on the platform where you run analysis, you can specify an options file in one of the following ways.

Command Line

At the command line (and in scripts), specify an options file as argument to the option `-options-file`.

For instance, instead of the command:

```
polyspace-bug-finder -sources file.c -compiler visual16.x -D _WIN32
        -code-behavior-specifications "Z:\utils\polyspace\forbiddenfunctions.xml"
```

Save this content:

```
-compiler visual16.x
-D _WIN32
-code-behavior-specifications "Z:\utils\polyspace\forbiddenfunctions.xml"
```

In a file `options.txt` in the path `Z:\utils\polyspace\` and shorten the command to:

```
polyspace-bug-finder -sources file.c -options-file "Z:\utils\polyspace\options.txt"
```

You can use options files with these Polyspace commands:

- `polyspace-bug-finder`
- `polyspace-bug-finder-server`
- `polyspace-bug-finder-access`
- `polyspace-code-prover`
- `polyspace-code-prover-server`

IDEs

If you run Polyspace as You Code using IDE extensions, you typically specify three groups of options differently:

Options Group	Extension Settings
Build options: You can extract build options from existing artifacts such as build commands and JSON compilation database. You can also collect all build options in an options file and specify the option file path 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 select checkers using a checkers selection wizard.	See “Setting Checkers in Polyspace as You Code”.
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”.

For more information on IDE extensions, see:

- “Configure Polyspace as You Code Extension in Visual Studio”
- “Configure Polyspace as You Code Extension in Visual Studio Code”
- “Configure Polyspace as You Code Plugin in Eclipse”

Polyspace User Interface

In the user interface of the Polyspace desktop products, you typically do not require an options file. Most options can be specified on the **Configuration** pane in the Polyspace user interface.

However, some options are available only at the command line and do not have a counterpart in the user interface. If you have to specify multiple command-line-only options, you can collect them in an options file, for instance `commandLineStyleOptions.txt`. On the **Configuration** pane, under the **Advanced Settings** node, specify the absolute path of the options file in the **Other** field:

```
-options-file C:\psconfig\commandLineStyleOptions.txt
```

Specifying Multiple Options Files

You can specify multiple options files in an analysis. For instance, at the command line, you can enter:

```
polyspace-bug-finder -sources file.c -options-file opts1.txt -options-file opts2.txt
```

When you specify multiple options files in an analysis, all options from the options files are appended to the analysis command. For instance, the preceding command has the same effect as using a single options file that places the content of `opts1.txt` above `opts2.txt`.

If an option appears in multiple files with conflicting arguments, the argument in the last options file prevails. For instance, in the preceding command, if `opts1.txt` contains:

```
-checkers all
-misra3 all
```

And `opts2.txt` contains:

```
-misra3 single-unit-rules
```

The analysis uses only the argument `single-unit-rules` for the option `-misra3`.

You can use this stacking of options files to override options. For instance, suppose you use a read-only options file that applies to your entire team but want to override some of the options in the file. You can override the options by using a second options file that you create and specifying your options file *after* the team-wide options file.

You can also specify the option `-options-file` within an options file and aggregate several options files in this way.

See Also

`-options-file`

Related Examples

- “Run Polyspace Analysis from Command Line” on page 4-2
- “Run Polyspace Code Prover on Server and Upload Results to Web Interface”
- “Complete List of Polyspace Code Prover Analysis Options”

Configure Target and Compiler Options

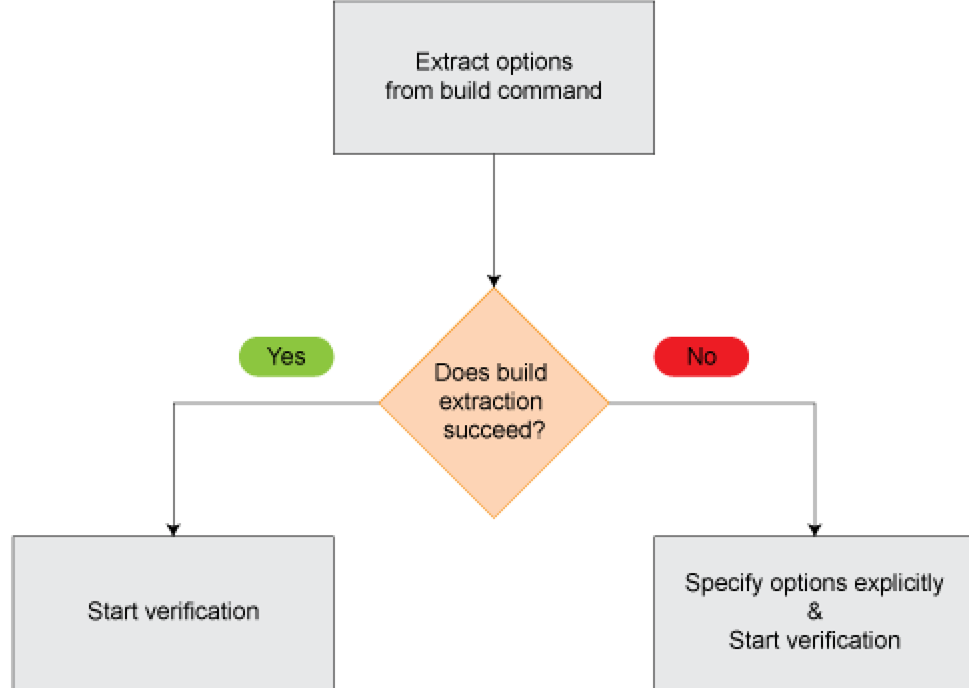
Specify Target Environment and Compiler Behavior

Before verification, specify your source code language (C or C++), target processor, and the compiler that you use for building your code. In certain cases, to emulate your compiler behavior, you might have to specify additional options.

Using your specification, the verification determines the sizes of fundamental types, considers certain macros as defined, and interprets compiler-specific extensions of the Standard. If the options do not correspond to your run-time environment, you can encounter:

- Compilation errors
- Verification results that might not apply to your target

If you use a build command such as `gmake` to build your code and the build command meets certain restrictions, you can extract the options from the build command after executing the command. Otherwise, specify the options explicitly.



Extract Options from Build Command

If you use build automation scripts to build your source code, you can set up a Polyspace project from your scripts. The options associated with your compiler are specified in that project.

In the Polyspace desktop products, for information on how to trace your build command from the:

- Polyspace user interface, see "Add Source Files for Analysis in Polyspace Desktop User Interface" on page 2-2.
- DOS or UNIX command line, see `polyspace-configure`.

- MATLAB command line, see `polyspaceConfigure`.

In the Polyspace server products, for information on how to trace your build command, see "Create Polyspace Analysis Configuration from Build Command (Makefile)" on page 13-21.

For Polyspace project creation, your build automation script (makefile) must meet certain requirements. See "Requirements for Project Creation from Build Systems" on page 13-23.

Specify Options Explicitly

If you cannot trace your build command and therefore manually create a project, you have to specify the options explicitly.

- In the user interface of the Polyspace desktop products, select a project configuration. On the **Configuration** pane, select **Target & Compiler**. Specify the options.
- At the DOS or UNIX command line, specify flags with the `polyspace-bug-finder`, `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server` command.
- At the MATLAB command line, specify arguments with the `polyspaceBugFinder`, `polyspaceCodeProver`, `polyspaceBugFinderServer` or `polyspaceCodeProverServer` function.

Specify the options in this order.

- Required options:
  - `Source code language (-lang)`: If all files have the same extension `.c` or `.cpp`, the verification uses the extension to determine the source code language. Otherwise, explicitly specify the option.
  - `Compiler (-compiler)`: Select the compiler that you use for building your source code. If you cannot find your compiler, use an option that closely matches your compiler.
  - `Target processor type (-target)`: Specify the target processor on which you intend to execute your code. For some processors, you can change the default specifications. For instance, for the processor `hc08`, you can change the size of types `double` and `long double` from 32 to 64 bits.

    If you cannot find your target processor, you can create your own target and specify the sizes of fundamental types, default signedness of `char`, and endianness of the target machine. See `Generic target options`.
- Language-specific options:
  - `C standard version (-c-version)`: The default C language standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C99 standard. Specify an earlier standard such as C90 or a later standard such as C11.
  - `C++ standard version (-cpp-version)`: The default C++ language standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C++03 standard. Specify later standards such as C++11 or C++14.
- Compiler-specific options:

  Whether these options are available or not depends on your specification for `Compiler (-compiler)`. For instance, if you select a `visual` compiler, the option `Pack alignment value`

`(-pack-alignment-value)` is available. Using the option, you emulate the compiler option `/Zp` that you use in Visual Studio.

For all compiler-specific options, see “Target and Compiler”.

- Advanced options:

  Using these options, you can modify the verification results. For instance, if you use the option `Division round down (-div-round-down)`, the verification considers that quotients from division or modulus of negative numbers are rounded down. Use these options only if you use similar options when compiling your code.

  For all advanced options, see “Target and Compiler”.
- Compiler header files:

  If you specify the `diab`, `tasking` or `greenhills` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis” on page 13-19.

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 13-19.

See Also

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

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 13-5
- “Provide Standard Library Headers for Polyspace Analysis” on page 13-19

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, C17

  The default standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C99 standard. To change the language standard, use the option `C standard version (-c-version)`.
- **C++**: C++03, C++11, C++14

  The default standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C++03 standard. To change the language standard, use the option `C++ standard version (-cpp-version)`.

Default Language Standard

The default language standard depends on your specification for the option `Compiler (-compiler)`.

GCC compilers

Compiler	C Standard	C++ Standard
`gnu3.4`, `gnu4.6`, `gnu4.7`, `gnu4.8`, `gnu4.9`	C99	C++03
`gnu5.x`	C11	C++03
`gnu6.x`	C11	C++14
`gnu7.x`	C11	C++14
`gnu8.x`	C17	C++14
`gnu9.x`	C17	C++14
`gnu10.x`	C17	C++14
`gnu11.x`	C17	C++17
`gnu12.x`	C17	C++17

Clang compilers

Compiler	C Standard	C++ Standard
`clang3.x`	C99	C++03 The analysis accepts some C++11 extensions.
`clang4.x`	C99	C++03 The analysis accepts C++14 extensions.
`clang5.x`	C99	C++03 The analysis accepts C++14 extensions.
`clang6.x`	C99	C++14
`clang7.x`	C99	C++14
`clang8.x`	C99	C++14
`clang9.x`	C99	C++14
`clang10.x`	C99	C++14
`clang11.x`	C17	C++14
`clang12.x`	C17	C++14
`clang13.x`	C17	C++14

Visual Studio compilers

Compiler	C Standard	C++ Standard

`visual9.0`
`visual10.0`
`visual11.0`
`visual12.0`
	C99	C++03

`visual14.0`	C99	C++14
`visual15.x`	C99	C++14
`visual16.x`	C99	C++14

Other Compilers

Compiler	C Standard	C++ Standard
`armcc`	C99	C++03
`armclang`	C11	C++03
`codewarrior`	C99	C++03
`cosmic`	C99	Not supported

Compiler	C Standard	C++ Standard
diab	C99	C++03
generic	C99	C++03
greenhills	C99	C++03
iar	C99	C++03
iar-ew	C99	C++03
intel	C11	C++14
keil	C99	C++03
microchip	C99	Not supported
renesas	C99	C++03
tasking	C99	C++03
ti	C99	C++03

See Also

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

More About

- “C11 Language Elements Supported in Polyspace” on page 13-8
- “C++11 Language Elements Supported in Polyspace” on page 13-9
- “C++14 Language Elements Supported in Polyspace” on page 13-12
- “C++17 Language Elements Supported in Polyspace” on page 13-15

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
`alignas` and `alignof` convenience macros	Yes
`aligned_alloc` function	Yes
`noreturn` convenience macros	Yes
Generic selection	Yes
Thread support library (`threads.h`)	Yes
Atomic operations library (`stdatomic.h`)	Yes
Atomic types with `_Atomic`	Yes. If you use the Clang compiler, see limitations book for limitations on atomic data types. See “Limitations of Polyspace Verification”.
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, `strcpy_s` is an alternative to `strcpy` 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 `errno_t` and `rsize_t` `typedef`-s	No. If you see compilation errors from use of these `typedef`-s, explicitly specify the path to your compiler headers. See “Provide Standard Library Headers for Polyspace Analysis” on page 13-19.
`quick_exit` and `at_quick_exit`	Yes. In Bug Finder, functions registered with `at_quick_exit` appear as uncalled.
`CMPLX`, `CMPLXF` and `CMPLXL` macros	Yes

See Also

`C standard version (-c-version)`

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 13-5

C++11 Language Elements Supported in Polyspace

This table provides a partial list of C++ language elements that have been introduced since C++11 and its corresponding Polyspace support. If your code contains nonsupported constructions, Polyspace reports a compilation error.

C++11 Std Ref	Description	Supported
C++2011-DR226	Default template arguments for function templates	Yes
C++2011-DR339	Solving the SFINAE problem for expressions	Yes
C++2011-N1610	Initialization of class objects by rvalues	Yes
C++2011-N1653	C99 preprocessor	Yes
C++2011-N1720	Static assertions	Yes
C++2011-N1737	Multi-declarator auto	Yes
C++2011-N1757	Right angle brackets	Yes
C++2011-N1791	Extended friend declarations	No
C++2011-N1811	long long	Yes
C++2011-N1984	auto-typed variables	Yes
C++2011-N1986	Delegating constructors	Yes
C++2011-N1987	Extern templates	Yes
C++2011-N1988	Extended integral types	Yes
C++2011-N2118	Rvalue references	Yes
C++2011-N2170	Universal character name literals	Yes
C++2011-N2179	Concurrency: Propagating exceptions	No
C++2011-N2235	Generalized constant expressions	Yes
C++2011-N2239	Concurrency: Sequence points	No new syntax/ keyword is introduced and therefore does not affect Polyspace support for C++11.
C++2011-N2242	Variadic templates	Yes
C++2011-N2249	New character types	Yes
C++2011-N2253	Extending sizeof	Yes
C++2011-N2258	Template aliases	Yes
C++2011-N2340	__func__ predefined identifier	Yes
C++2011-N2341	Alignment support	Yes
C++2011-N2342	Standard Layout Types	Yes
C++2011-N2343	Declared type of an expression	Yes
C++2011-N2346	Defaulted and deleted functions	Yes
C++2011-N2347	Strongly typed enums	Yes

C++11 Std Ref	Description	Supported
C++2011-N2427	Concurrency: Atomic operations	No
C++2011-N2429	Concurrency: Memory model	No new syntax/ keyword is introduced and therefore does not affect Polyspace support for C++11.
C++2011-N2431	Null pointer constant	Yes
C++2011-N2437	Explicit conversion operators	Yes
C++2011-N2439	Rvalue references for *this	Yes
C++2011-N2440	Concurrency: Abandoning a process and at_quick_exit	Yes
C++2011-N2442	Unicode string literals	Yes
C++2011-N2442	Raw string literals	Yes
C++2011-N2535	Inline namespaces	Yes
C++2011-N2540	Inheriting constructors	Yes
C++2011-N2541	New function declarator syntax	Yes
C++2011-N2544	Unrestricted unions	Yes
C++2011-N2546	Removal of auto as a storage-class specifier	Yes
C++2011-N2547	Concurrency: Allow atomics use in signal handlers	No
C++2011-N2555	Extending variadic template template parameters	Yes
C++2011-N2657	Local and unnamed types as template arguments	Yes
C++2011-N2659	Concurrency: Thread-local storage	No
C++2011-N2660	Concurrency: Dynamic initialization and destruction with concurrency	Yes
C++2011-N2664	Concurrency: Data-dependency ordering: atomics and memory model	No
C++2011-N2672	Initializer lists	Yes
C++2011-N2748	Concurrency: Strong Compare and Exchange	No
C++2011-N2752	Concurrency: Bidirectional Fences	No
C++2011-N2756	Nonstatic data member initializers	Yes
C++2011-N2761	Generalized attributes	Yes
C++2011-N2764	Forward declarations for enums	Yes
C++2011-N2765	User-defined literals	Yes
C++2011-N2927	New wording for C++0x lambdas	Yes
C++2011-N2928	Explicit virtual overrides	Yes
C++2011-N2930	Range-based for	Yes
C++2011-N3050	Allowing move constructors to throw [noexcept]	Yes
C++2011-N3053	Defining move special member functions	Yes

C++11 Std Ref	Description	Supported
C++2011-N3276	decltype and call expressions	Yes

See Also
`C++ standard version (-cpp-version)`

More About
- "C/C++ Language Standard Used in Polyspace Analysis" on page 13-5
- "C++14 Language Elements Supported in Polyspace" on page 13-12
- "C++17 Language Elements Supported in Polyspace" on page 13-15

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 `delete` or `switch` 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 `std::result_of`	Yes
C++2014-N3472	Binary literals, for instance, `0b100`.	Yes
C++2014-N3545	`operator()` in `integral_constant` template of `constexpr` type	Yes
C++2014-N3637	Relation between `std::async` and destructor of `std::future`	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 (`h`, `min`, `s`, `ms`, `us`, `ns`) and strings (`s`)	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: • Using `auto` 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 `constexpr` functions.	Yes

C++2014-N3653	Initialization of aggregate classes with fewer initializers than members
	
	For instance, this initialization has fewer initializers than members. The member `c` is initialized with the value 0 and `d` is initialized with the value `s`.
	
	`struct S {`
	`   int a;`
	`   const char* b;`
	`   int c;`
	`   int d = b[a];};`
	`S ss = { 1, "asdf" };`
		Yes

C++2014-N3654	`std::quoted`	Yes
C++2014-N3656	`std::make_unique`	Yes
C++2014-N3658	`std::integer_sequence`	Yes
C++2014-N3658	`std::shared_lock`	No. The use of `std::shared_lock` does not cause compilation errors but the construct is not semantically supported.
C++2014-N3664	Calling `new` and `delete` operators in batches.	This C++14 feature clarifies how successive calls to the `new` operator are implemented. No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3668	`std::exchange`	Partially supported.
C++2014-N3670	Using `std::get` with a data type to get one element in an `std::tuple` (provided there is only one element of the type in the tuple)	Yes
C++2014-N3671	Overloads for `std::equal`, `std::mismatch` and `std::is_permutation` function templates that accept two separate ranges	Yes
C++2014-N3733	Removal of `std::gets` from `<cstdio>`	Yes

C++14 Std Ref	Description	Supported
C++2014-N3776	Wording change for destructor of `std::future`	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3779	`std::complex` literals representing pure imaginary numbers with suffix `i`, `if` or `il`	Yes
C++2014-N3781	Use of single quotation mark as digit separator, for instance, `1'000`.	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 `rand()`	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3927	Lock-free executions	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.

See Also

`C++ standard version (-cpp-version)`

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 13-5
- “C++11 Language Elements Supported in Polyspace” on page 13-9
- “C++17 Language Elements Supported in Polyspace” on page 13-15

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	`std::string-view`: Observe the content of an `std::string` object without owning the resource	Yes
C++2017-N3922	• When used in copy-list-initialization, `auto` deduces the type to be an `std::initializer_list` if the elements of the initializer list have an identical type. Otherwise, the `auto` deduction is ill-formed. • When using direct list-initialization with a braced initializer list containing a single element, `auto` deduces the type from that element. • When using direct list-initialization with a braced initializer list containing more than a single element, `auto` deduction of type is ill-formed.	Yes
C++2017-N3928	The `static_assert` declaration no longer requires a second argument. Invoking `static_assert` with no message is now allowed: `static_assert(N > 0);`	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 `class` keyword. In C++17, you can use `typename` when declaring template-template parameters , such as: `template <template <typename> typename Tmpl> struct X;`	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:
	• `namespace base::derived{`
	`//..`
	`}`
	• `namespace {`
	`    namespace derived{`
	`        //...`
	`    }`
	`}`
		Yes

C++17 Std Ref	Description	Supported
C++2017-N4259	The function `std::uncaught_exceptions` is introduced in C++17, which returns the number of exceptions in your code that are not handled. The function `std:uncaught_exception`, which returns a Boolean value, is deprecated.	Yes
C++2017-N4266	Starting in C++17, namespaces and enumerators can be annotated with attributes to allows clearer communication of developer intention.	Yes
C++2017-N4267	Starting in C++17, the prefix `u8` 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 untimed `std::shared_mutex`	The use of `std::shared_mutex` does not cause a compilation error. Polyspace does not support sharing mutex objects by using `std::shared_mutex`.
C++2017-P0001R1	Remove the use of the `register` keyword	Yes
C++2017-P0002R1	Remove `operator++(bool)`	Yes
C++2017-P0003R5	Remove deprecated exception specifications by using `throw(<>)`	Bug Finder removes the exception specification specified by using `throw()` statements. Code Prover raises a compilation error when `throw()` 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 `*this` 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 `__has_include` 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	`std::variant`	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	`constexpr` Lambda expressions	Partially supported
C++2017-P018R0	Differing begin and end types in range-based `for` loops	Yes
C++2017-P0188R1	`[[fallthrough]]` 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 *template-argument* is a valid match for a *template-parameter*	Partially supported

See Also

`C++ standard version (-cpp-version)`

More About

- "C/C++ Language Standard Used in Polyspace Analysis" on page 13-5
- "C++11 Language Elements Supported in Polyspace" on page 13-9
- "C++14 Language Elements Supported in Polyspace" on page 13-12

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.

If you create a Polyspace project or options file from your build command using `polyspace-configure`, the header paths are automatically added to this project or options file. Otherwise, you have to explicitly add these paths. This topic shows how to locate the standard library headers from your compiler. The code examples cause a compilation error that shows the location of the headers.

- To locate the folder containing your C compiler system headers, compile this C code by using your compilation toolchain:

  ```
  float fopen(float f);
  #include <stdio.h>
  ```

  The code does not compile because the `fopen` declaration conflicts with the declaration inside `stdio.h`. The compilation error shows the location of your compiler implementation of `stdio.h`. Your C standard library headers are all likely to be in that folder.
- To locate the folder containing your C++ compiler system headers, compile this C++ code by using your compilation toolchain:

  ```
  namespace std {
      float cin;
  }
  #include <iostream>
  ```

  The code does not compile because the `cin` declaration conflicts with the declaration inside `iostream.h`. The compilation error shows the location of your compiler implementation of `iostream.h`. Your C++ standard library headers are all likely to be in that folder.

After you locate the path to your compiler's header files, specify the path for the Polyspace analysis. For C++ code, specify the paths to both your C and C++ headers.

- In the user interface (Polyspace desktop products), add the folder to your project.

  For more information, see “Add Source Files for Analysis in Polyspace Desktop User Interface” on page 2-2.
- At the command line, use the flag `-I` with one of these commands:
  - `polyspace-bug-finder`
  - `polyspace-bug-finder-server`
  - `polyspace-code-prover`
  - `polyspace-code-prover-server`

  For more information, see `-I`.

See Also

More About

- “Fix Errors from Use of Polyspace Header Files” on page 34-65

Create Polyspace Analysis Configuration from Build Command (Makefile)

To run Polyspace with scripts at regular intervals, for instance, on a server during continuous integration, you must configure all analysis options beforehand so that the analysis completes without errors. These options must be updated as necessary to keep up with new code submissions. If you use existing artifacts such as a build command (makefile) to build new code submissions, you can reuse the build command to configure a Polyspace analysis and stay updated with new submissions. With the `polyspace-configure` command, you can monitor the execution of a build command and create an options file for analysis with Polyspace.

This topic shows a simple tutorial illustrating how to create an options file from a build command and use the file for the subsequent analysis. The topic uses a Linux makefile and the GCC compiler, but you can adapt the commands to other operating systems such as Windows and other compilers such as Microsoft Visual Studio.

1. Copy the demo source files from *`polyspaceserverroot`*`\polyspace\examples\cxx\Bug_Finder_Example\sources` to a folder with write permissions. Here, *`polyspaceserverroot`* is the root installation folder of the Polyspace server products, for instance, `C:\Program Files\Polyspace Server\R2019a`.
2. Create a simple makefile that compiles the demo source files. Save the makefile in the same folder as the source files.

   For instance, create a file named `makefile` and add this content:

   ```
   CC := gcc
   SOURCES := $(wildcard *.c)

   all: $(CC) -c $(SOURCES)
   ```

   Check that the makefile builds the source files successfully. Open a command terminal, navigate to the folder (using `cd`) and enter:

   ```
   make -B
   ```

   The `make` command should complete execution without errors.

   The `-B` option ensures that all targets in the makefile are built. Typically, build commands such as `make` are set up to only build sources that have changed since the previous build. However, `polyspace-configure` requires a full build to determine which sources to add to a Polyspace project or options file.
3. Trace the build command with `polyspace-configure` and create an options file `compile_opts`.

   ```
   polyspace-configure -output-options-file compile_opts make -B
   ```
4. Create a second options file with additional options. For instance, create a file `run_opts` with this content:

   ```
   -checkers numerical
   -report-template BugFinder
   -output-format pdf
   ```

The options run all numerical checkers in Bug Finder and creates a PDF report after analysis using the `BugFinder` template.

5. Run a Bug Finder analysis with the two options files: `compile_opts` created from your build command and `run_opts` created manually.

   Polyspace Bug Finder:

   ```
   polyspace-bug-finder -options-file compile_opts -options-file run_opts
   ```

   Polyspace Bug Finder Server:

   ```
   polyspace-bug-finder-server -options-file compile_opts -options-file run_opts
   ```

   The analysis should complete without errors. You can open the results in the Polyspace user interface or upload the results to the Polyspace Access web interface (using the `polyspace-access` command).

   To run Code Prover instead of Bug Finder, use the `polyspace-code-prover-server` command instead of the `polyspace-bug-finder-server` command.

You can run a similar analysis using MATLAB scripts. Replace `polyspace-bug-finder` with the `polyspaceBugFinder` function and `polyspace-configure` with the function `polyspaceConfigure`.

See Also

`polyspace-configure` | `polyspace-code-prover-server`

See Also

More About

- "Specify Target Environment and Compiler Behavior" on page 13-2
- "Select Files for Polyspace Analysis Using Pattern Matching" on page 4-11
- "Modularize Polyspace Analysis by Using Build Command" on page 4-5

Requirements for Project Creation from Build Systems

For automatic project creation from build systems, your build commands or makefiles must meet certain requirements.

Compiler Requirements

- Your compiler must be called locally.

  If you use a compiler cache such as `ccache` or a distributed build system such as `distmake`, the software cannot trace your build. You must deactivate them.
- Your compiler must perform a clean build.

  If your compiler performs only an incremental build, use appropriate options to build all your source files. For example, if you use `gmake`, append the `-B` or `-W` *`makefileName`* option to force a clean build. For the list of options allowed with the GNU® `make`, see make options.
- Your compiler configuration must be available to Polyspace. The compilers currently supported include the following:

  - Arm Keil.
  - ARM® v5. See also `ARM v5 Compiler (-compiler armcc)` .
  - ARM v6. See also `ARM v6 Compiler (-compiler armclang)`.
  - Clang. For a list of supported versions, see “Clang Compilers”.
  - Cosmic. See also `Cosmic Compiler (-compiler cosmic)`.
  - Wind River® Diab. See also `Diab Compiler (-compiler diab)`.
  - Green Hills®. See also `Green Hills Compiler (-compiler greenhills)`.
  - GNU C/C++. For a list of supported versions, see “GCC Compilers”.
  - IAR Embedded Workbench. See also `IAR Embedded Workbench Compiler (-compiler iar-ew)` .
  - IAR systems.
  - Intel® C++ Compiler Classic (icc/icl) compiler. See also `Intel C++ Compiler Classic (icc/icl) (-compiler intel)`.
  - Microsoft Visual C++®. For a list of supported versions, see “Visual Studio Compilers”.
  - MPLAB XC8 C. See also `MPLAB XC8 C Compiler (-compiler microchip)`.
  - NXP CodeWarrior®. See also `NXP CodeWarrior Compiler (-compiler codewarrior)`.
  - Renesas®. See also `Renesas Compiler (-compiler renesas)`.
  - Altium® Tasking. See also `TASKING Compiler (-compiler tasking)`.
  - Texas Instruments®. See also `Texas Instruments Compiler (-compiler ti)`.
  - tcc - Tiny C Compiler

  If your compiler configuration is not available to Polyspace:

  - Write a compiler configuration file for your compiler in a specific format. For more information, see “Create Polyspace Projects from Build Systems That Use Unsupported Compilers” on page 34-25.

  - Contact MathWorks Technical Support. For more information, see "Contact Technical Support About Issues with Running Polyspace" on page 34-18.
- If you build your code in Cygwin™, use versions 2.x or 3.x of Cygwin for Polyspace project creation from your build system (for instance, Cygwin version 2.10 or 3.0).
- With the TASKING compiler, if you use an alternative sfr file with extension `.asfr`, Polyspace might not be able to locate your file. If you encounter an error, explicitly `#include` your `.asfr` file in the preprocessed code using the option `Include (-include)`.

  Typically, you use the statement `#include __SFRFILE__(__CPU__)` along with the compiler option `--alternative-sfr-file` to specify an alternative sfr file. The path to the file is typically *`Tasking_C166_INSTALL_DIR`*`\include\sfr\reg`*`CPUNAME`*`.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 34-32.
- Your build command must not contain lines where several sources are compiled in a single line using wildcard characters, for instance:

  ```
  cl.exe *.c
  ```
- If you use statically linked libraries, Polyspace cannot trace your build. In Linux, you can install the full Linux Standard Base (LSB) package to allow dynamic linking. For example, on Debian® systems, install LSB with the command `apt-get install lsb`.
- Your build command must not use aliases.

  The `alias` command is used in Linux to create an alternate name for commands. If your build command uses those alternate names, Polyspace cannot recognize them.
- Your build process must not use the `LD_PRELOAD` mechanism.
- Your build command must be executable completely on the current machine and must not require privileges of another user.

  If your build uses `sudo` to change user privileges or `ssh` to remotely log in to another machine, Polyspace cannot trace your build.
- If your build command uses redirection with the `>` or `|` character, the redirection occurs after Polyspace traces the command. Therefore, Polyspace does not handle the redirection.

  For example, if your command occurs as

  ```
  command1 | command2
  ```

  And you enter

```
polyspace-configure command1 | command2
```

When tracing the build, Polyspace traces the first command only.

- If the System Integrity Protection (SIP) feature is active on the operating system macOS El Capitan (10.11) or a later macOS version, you can use `polyspace-configure` to generate a project or build options file only if your build system supports the generation of a compilation database file that you pass to `polyspace-configure`. See “Create Polyspace Options File from JSON Compilation Database”. If your build system does not support the generation of compilation database file and the SIP feature is enabled, Polyspace cannot trace your build command. Alternatively, before tracing your build command, disable the SIP feature. You can reenable this feature after tracing the build command.

  Similar considerations apply to other security applications such as security-related products from CylanceProtect, Avecto and Tanium.
- If your computer hibernates during the build process, Polyspace might not be able to trace your build.
- When creating projects from build commands in the Polyspace User Interface, you might encounter errors such as `libcurl.so.4: version 'CURL_OPENSSL_3' not found`. In such cases, create the Polyspace project by using the command `polyspace-configure` in the system command line interface, using the build command as the argument. See `polyspace-configure`.

Note Your environment variables are preserved when Polyspace traces your build command.

See Also

`polyspace-configure`

Related Examples

- “Add Source Files for Analysis in Polyspace Desktop User Interface” on page 2-2
- “Create Polyspace Analysis Configuration from Build Command (Makefile)” on page 13-21

Supported Keil or IAR Language Extensions

Polyspace analysis can interpret a subset of common C/C++ language constructs and extended keywords by default. For compiler-specific keywords, you must specify your choice of compiler. If you specify `keil` or `iar` for `Compiler (-compiler)`, the Polyspace verification allows language extensions specific to the Keil or IAR compilers.

Special Function Register Data Type

Embedded control applications frequently read and write port data, set timer registers, and read input captures. To deal with these requirements without using assembly language, some microprocessor compilers define special data types such as `sfr` and `sbit`. Typical declarations are:

```
sfr A0 = 0x80;
sfr A1 = 0x81;
sfr ADCUP = 0xDE;
sbit EI = 0x80;
```

The declarations reside in header files such as `regxx.h` for the basic `80Cxxx` micro processor. The declarations customize the compiler to the target processor.

You access a register or a port by using the `sfr` and `sbit` data as follows. However, these data types are not part of the C99 Standard.

```
int status,P0;

void main (void) {
  ADCUP = 0x08; /* Write data to register */
  A1 = 0xFF; /* Write data to Port */
  status = P0; /* Read data from Port */
  EI = 1; /* Set a bit (enable all interrupts) */
}
```

To analyze this type of code, use these options:

- `Compiler (-compiler)`: Specify `keil` or `iar`.
- `Sfr type support (-sfr-types)`: Specify the data type and size in bits.

For example, depending on how you define the `sbit` data type, you use these options:

- `sbit ADST = ADCUP^7;`

  Use options: `-compiler keil -sfr-type sfr=8`
- `sbit ADST = ADCUP.7;`

  Use options: `-compiler iar -sfr-type sfr=8`

The analysis then supports the Keil or IAR language extensions even if some structures, keywords, and syntax are not part of the C99 standard.

Keywords Removed During Preprocessing

Once you specify the Keil or IAR compiler, the analysis recognizes compiler-specific keywords in your code. If a keyword is not relevant for the analysis, it is removed from the source code during preprocessing.

If you disable the keyword and use it as an identifier instead, you can encounter a compilation error when you compile your code with Polyspace. See “Fix Polyspace Compilation Errors Related to Keil or IAR Compiler” on page 34-51.

These keywords are removed during preprocessing:

- Keil: `bdata`, `far`, `idata`, `huge`, `sdata`
- IAR: `saddr`, `reentrant`, `reentrant_idata`, `non_banked`, `plm`, `bdata`, `idata`, `pdata`, `code`, `xdata`, `xhuge`, `interrupt`, `__interrupt`, `__intrinsic`

  The `data` keyword is not removed.

Remove or Replace Keywords Before Compilation

The Polyspace compiler strictly follows the ANSI® C99 Standard (ISO/IEC 9899:1999). If your compiler allows deviation from the Standard, the Polyspace compilation using default options cannot emulate your compiler. For instance, your compiler can allow certain non-ANSI keyword, which Polyspace does not recognize by default.

To emulate your compiler closely, you specify the Target & Compiler options. If you still get compilation errors from unrecognized keywords, you can remove or replace them only for the purposes of verification. The option `Preprocessor definitions (-D)` allows you to make simple substitutions. For complex substitutions, for instance to remove a group of space-separated keywords such as a function attribute, use the option `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

Remove Unrecognized Keywords

You can remove unsupported keywords from your code for the purposes of analysis. For instance, follow these steps to remove the `far` and `0x` keyword from your code (`0x` precedes an absolute address).

1 Save the following template as `C:\Polyspace\myTpl.pl`.

Content of myTpl.pl

```
#!/usr/bin/perl

##############################################################
# Post Processing template script
#
##############################################################
# Usage from GUI:
#
# 1) Linux: /usr/bin/perl PostProcessingTemplate.pl
# 2) Windows: polyspaceroot\sys\perl\win32\bin\perl.exe <pathtoscript>\
# PostProcessingTemplate.pl
#
##############################################################

$version = 0.1;

$INFILE = STDIN;
$OUTFILE = STDOUT;

while (<$INFILE>)
{

    # Remove far keyword
    s/far//;

    # Remove "@ 0xFE1" address constructs
    s/\@\s0x[A-F0-9]*//g;

    # Remove "@0xFE1" address constructs
    s/\@0x[A-F0-9]*//g;

    # Remove "@ ((unsigned)&LATD*8)+2" type constructs
    s/\@\s\(\(unsigned\)\&[A-Z0-9]+\*8\)\+\d//g;

    # DON'T DELETE LINE BELOW: Print the current processed line
    print $OUTFILE $_;
}
```

For reference, see a summary of Perl regular expressions.

Perl Regular Expressions

```
#########################################################
# Metacharacter What it matches
```

```
#########################################################
# Single Characters
# . Any character except newline
# [a-z0-9] Any single character in the set
# [^a-z0-9] Any character not in set
# \d A digit same as
# \D A non digit same as [^0-9]
# \w An Alphanumeric (word) character
# \W Non Alphanumeric (non-word) character
#
# Whitespace Characters
# \s Whitespace character
# \S Non-whitespace character
# \n newline
# \r return
# \t tab
# \f formfeed
# \b backspace
#
# Anchored Characters
# \B word boundary when no inside []
# \B non-word boundary
# ^ Matches to beginning of line
# $ Matches to end of line
#
# Repeated Characters
# x? 0 or 1 occurrence of x
# x* 0 or more x's
# x+ 1 or more x's
# x{m,n} Matches at least m x's and no more than n x's
# abc All of abc respectively
# to|be|great One of "to", "be" or "great"
#
# Remembered Characters
# (string) Used for back referencing see below
# \1 or $1 First set of parentheses
# \2 or $2 First second of parentheses
# \3 or $3 First third of parentheses
##########################################################
# Back referencing
#
# e.g. swap first two words around on a line
# red cat -> cat red
# s/(\w+) (\w+)/$2 $1/;
#
##########################################################
```

2. On the **Configuration** pane, select **Environment Settings**.
3. To the right of **Command/script to apply to preprocessed files**, click .
4. Use the Open File dialog box to navigate to `C:\Polyspace`.
5. In the **File name** field, enter `myTpl.pl`.
6. Click **Open**. You see `C:\Polyspace\myTpl.pl` in the **Command/script to apply to preprocessed files** field.

Remove Unrecognized Function Attributes

You can remove unsupported function attributes from your code for the purposes of analysis.

If you run verification on this code specifying a generic compiler, you can see compilation errors from the `noreturn` attribute. The code compiles using a GNU compiler.

```
void fatal () __attribute__ ((noreturn));

void fatal (/* ... */)
{
    /* ... */ /* Print error message. */ /* ... */
    exit (1);
}
```

If the software does not recognize an attribute and the attribute does not affect the code analysis, you can remove it from your code for the purposes of verification. For instance, you can use this Perl script to remove the `noreturn` attribute.

```
while ($line = <STDIN>)
{

# __attribute__ ((noreturn))

  # Remove far keyword
  $line =~ s/__attribute__\ \(\(noreturn\)\)//g;

  # Print the current processed line to STDOUT
  print $line;
}
```

Specify the script using the option `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

See Also

Polyspace Analysis Options
`Command/script to apply to preprocessed files (-post-preprocessing-command)` | `Preprocessor definitions (-D)`

Related Examples

- "Troubleshoot Compilation Errors"

Gather Compilation Options Efficiently

Polyspace verification can sometimes stop in the compilation or linking phase due to the following reasons:

- The Polyspace compiler strictly follows a C or C++ Standard (depending on your choice of compiler). See “C/C++ Language Standard Used in Polyspace Analysis” on page 13-5. If your compiler allows deviation from the Standard, the Polyspace compilation using default options cannot emulate your compiler.
- Your compiler declares standard library functions with argument or return types different from the standard types. Unless you also provide the function definition, for efficient verification, Polyspace uses its own definitions of standard library functions, which have the usual prototype. The mismatch in types causes a linking error.

You can easily work around the compilation and standard library function errors. To work around the errors, you typically specify certain analysis options. In some cases, you might have to add a few lines to your code. For instance:

- To emulate your compiler behavior more closely, you specify the Target & Compiler options. If you still face compilation errors, you might have to remove or replace certain unrecognized keywords using the option `Preprocessor definitions (-D)`. However, the option allows only simple substitution of a string with another string. For more complex replacements, you might have to add `#define` statements to your code.
- To avoid errors from stubbing standard library functions, you might have to `#define` certain Polyspace-specific macros so that Polyspace does not use its own definition of standard library functions.

Instead of adding these modifications to your original code, create a single `polyspace.h` file that contains all modifications. Use the option `Include (-include)` to force inclusion of the `polyspace.h` file in all source files under verification.

Benefits of this approach include:

- The error detection is much faster since it will be detected during compilation rather than in the link or subsequent phases.
- There will be no need to modify original source files.
- The file is automatically included as the very first file in the original `.c` files.
- The file is reusable for other projects developed under the same environment.

Example 13.1. Example

This is an example of a file that can be used with the option `Include (-include)`.

```
// The file may include (say) a standard include file implicitly
// included by the cross compiler

#include <stdlib.h>
#include "another_file.h"

// Workarounds for compilation errors
#define far
#define at(x)
```

```
// Workarounds for errors due to redefining standard library functions

#define POLYSPACE_NO_STANDARD_STUBS    // use this flag to prevent the
           //automatic stubbing of std functions
#define __polyspace_no_sscanf
#define __polyspace_no_fgetc
void sscanf(int, char, char, char, char, char);
void fgetc(void);
```

See Also

More About

- “Troubleshoot Compilation Errors”

Configure Inputs and Stubbing Options

Specify External Constraints for Polyspace Analysis

Polyspace products analyzes C/C++ code and checks for issues such as defects (bugs) or run-time errors. The analysis uses the code that you provide to make assumptions about items such as variable ranges and allowed buffer size for pointers. Sometimes the assumptions are broader than what you expect because:

- You have not provided the complete code. For example, you did not provide some of the function definitions.
- Some of the information about variables is available only at run time. For example, some variables in your code obtain values from the user at run time.

Because of these broad assumptions:

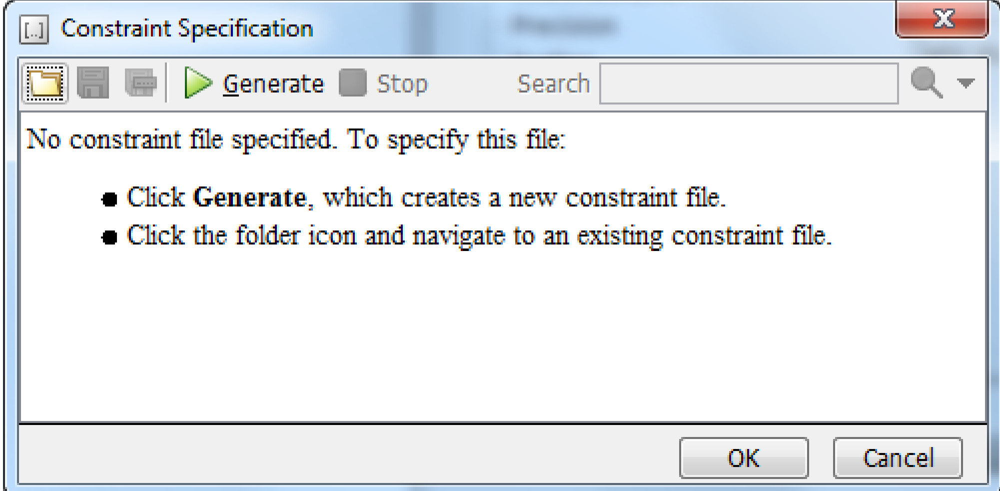
- Code Prover can consider more execution paths than those paths that occur at run time. If an operation fails along one of the execution paths, Polyspace places an orange check on the operation. If that execution path comes from an assumption that is too broad, the orange check might indicate a false positive.
- Bug Finder can sometimes produce false positives.

To reduce the number of such false positives, you can specify additional constraints on global variables, function inputs, and return values and modifiable arguments of stubbed functions. This example shows how to specify these external constraints (also known as data range specifications or DRS). You save the constraints as an XML file to use them for subsequent analyses. If your source code changes, you can update the previous constraints. You do not have to create a new constraint template.

Create Constraint Template

User Interface (Desktop Products Only)

1. Open the project configuration. On the **Configuration** pane, select **Inputs & Stubbing**.
2. To the right of **Constraint setup**, click the **Edit** button to open the **Constraint Specification** window.



3. In the Constraint Specification dialog box, create a blank constraint template. The template contains a list of all variables on which you can provide constraints. To create a new template, click Generate. The software compiles your project and creates a template. The new template is stored in a file `Module_number_Project_name_drs_template.xml` in your project folder.
4. Specify your constraints and save the template as an XML file. For more information, see "External Constraints for Polyspace Analysis" on page 14-6.
5. Click **OK**.

   You see the full path to the template XML file in the **Constraint setup** field. If you run an analysis, Polyspace uses this template for extracting variable constraints.

Command Line

Use the option `Constraint setup (-data-range-specifications)` to specify the constraints XML file.

To specify constraints in the XML file:

1. First, create a blank XML template. The template lists all global variables, function inputs and modifiable arguments and return values of stubbed functions without specifying any constraints on them.

   To create a blank template, run an analysis only up to the compilation phase. In Bug Finder, disable checking of defects. Use the option `Find defects (-checkers)`. In Code Prover, check for source compliance only. Use the argument `compile` for the option `Verification level (-to)`. After the analysis, a blank template XML `drs-template.xml` is created in the results folder.

   For C++ projects, to create a blank constraints template, you have to use the argument `cpp-normalize` for the option `Verification level (-to)`.
2. Edit the XML file to specify your constraints.

   For examples, see:

   - "Constrain Global Variable Range for Polyspace Analysis" on page 14-12
   - "Constrain Function Inputs for Polyspace Analysis" on page 14-14

Create Constraint Template from Code Prover Analysis Results

You can constrain variable ranges based on their expected range in real-world applications. For instance, if a variable represents vehicle speed, you can set a maximum possible value. You can also constrain variable ranges only if they cause too many orange checks from overapproximation.

A Code Prover analysis shows all global variables, function inputs and stubbed functions that lead to orange checks from possible overapproximation. You can constrain only these variables for a more precise analysis.

1. Open Code Prover results in the Polyspace user interface or Polyspace Access web interface.
2. Open the **Orange Sources** pane. Do one of the following:

- Select an orange check. If the software can trace an orange check to a root cause, a icon appears on the **Result Details** pane. Click this icon to open the **Orange Sources** pane.
- In the Polyspace user interface, select **Window** > **Show/Hide View** > **Orange Sources**. In the Polyspace Access web interface, select **Window** > **Orange Sources**.

  You see the full list of variables (function inputs or return values of stubbed functions) that can cause orange checks. Constrain the ranges of these variables.

In the details for individual orange checks, you often see a message similar to this:

```
If appropriate, applying DRS to stubbed function random_float in example.c
line 44 may remove this orange.
```

The message is an indication that the stubbed function is a possible source of the orange check. You can apply external constraints on the function to enforce more precise assumptions and possibly remove the orange check (in case it came from the broader assumptions).

Update Existing Template

With new code submissions, you might have to specify additional constraints. You can update an existing template to add global variables, function inputs and stubbed functions that come from the new code submissions.

Additionally, if you remove some variables or functions from your code, constraints on them are not applicable any more. Instead of regenerating a constraint template and respecifying the constraints, you can update an existing template and remove the variables that are not present in your code.

User Interface (Desktop Products Only)

1. On the **Configuration** pane, select **Inputs & Stubbing**.
2. Open the existing template in one of the following ways:
   - In the **Constraint setup** field, enter the path to the template XML file. Click **Edit**.
   - Click **Edit**. In the Constraint Specification dialog box, click the icon to navigate to your template file.
3. Click **Update**.
   a. Variables that are no longer present in your source code appear under the **Non Applicable** node. To remove an entry under the **Non Applicable** node or the node itself, right-click and select **Remove This Node**.
   b. Specify your new constraints for any of the other variables.

Command Line

In a continuous integration workflow, you can use the constraints XML file from the previous run. If new code submissions require additional constraints:

1. Specify constraints on variables from new code submissions in a constraints XML file. See Create Constraint Template: Command Line.
2. Merge the constraints XML file with the new constraints and the constraints XML file from the previous run.

Specify Constraints in Code

Specifying constraints outside your code allows for more precise analysis. However, you must use the code within the specified constraints because the constraints are *outside* your code. Otherwise, the results might not apply. For example, if you use function inputs outside your specified range, a run-time error can occur on an operation even though checks on the operation are green.

To specify constraints *inside* your code, you can use:

- Appropriate error handling tests in your code.

  Polyspace checks to determine if the errors can actually occur. If they do not occur, the test blocks appear as **Unreachable code**.
- The `unchecked_assert` macro. For example, to constrain a variable `var` in the range [0,10], you can use `assert(var >= 0 && var <=10);`.

  See “Constrain Variable Ranges for Polyspace Analysis Using Manual Stubs and Manual main() Function” on page 18-18.

See Also

`Constraint setup (-data-range-specifications)`

Related Examples

- “External Constraints for Polyspace Analysis” on page 14-6
- “Constrain Global Variable Range for Polyspace Analysis” on page 14-12
- “Constrain Function Inputs for Polyspace Analysis” on page 14-14
- “XML File Format for Polyspace Analysis Constraints” on page 14-17
- “Constrain Variable Ranges for Polyspace Analysis Using Manual Stubs and Manual main() Function” on page 18-18

External Constraints for Polyspace Analysis

Polyspace uses the code that you provide to make assumptions about items such as variable ranges and allowed buffer size for pointers. Sometimes the assumptions that Polyspace makes are broader than what you expect, which might result in Bug Finder false positives or more Code Prover orange checks. To reduce such false positives or orange checks, you can specify external constraints on:

- Global variables.
- User-defined functions.
- Stubbed functions.

For more information, see “Specify External Constraints for Polyspace Analysis” on page 14-2. For a partial list of limitations, see “Constraint Specification Limitations”.

Effect of External Constraints

Consider the following function:

```
int getFlooredNumber(int total, int size) {
    return total/size;
}
```

Since the input `size` is unknown, if you analyze this function:

- With Polyspace Code Prover, you see an orange **Division by zero** check. The orange check indicates that Code Prover suspects a possible division by zero error, but this error does not occur on all execution paths.
- With Polyspace Bug Finder, if you use the option `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`, you see an **Integer division by zero** defect along with one possible input value leading to the defect (in this case, a `size` of 0).

  In more sophisticated examples, for instance, if the division occurs inside a condition, a defect appears from unknown inputs even without using the option.

In both cases, the analysis determines possible values of the input variable `size` from its data type. Since the variable `size` has data type `int`, on targets where `int` has a size of 32 bits, the variable is assumed to have values in the range $[-2^{31}, 2^{31}-1]$.

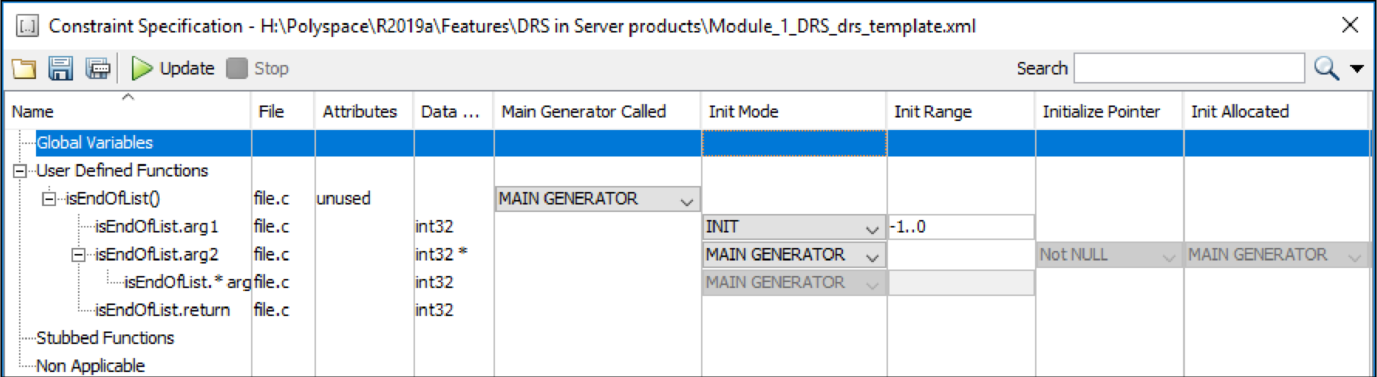
If you know that an input has values only within a certain range, you can specify an external constraint on the input (also known as Data Range Specification or DRS). For instance, in the above example, if you specify a range on `size` that excludes zero:

- Code Prover no longer shows the orange **Division by zero** check.
- Bug Finder no longer shows the **Integer division by zero** defect.

You can specify external constraints to emulate design constraints that live outside your code. For instance, if an input variable represents a physical quantity such as vehicle speed, you can constrain the variable values to speeds for which the vehicle is designed.

Constraint Specification

In the user interface of the Polyspace desktop products, you can specify the constraints through a **Constraint Specification** window. The constraints are saved in an XML file that you can reuse for other projects.



This table describes the various columns in the **Constraint Specification** window. If you directly edit the constraint XML file to specify a constraint (for instance, in the Polyspace Server products), this table also shows the correspondence between columns in the user interface and entries in the XML file. The XML entry highlighted in bold appears in the corresponding column of the **Constraint Specification** window.

Column	Settings
Name	Displays the list of variables and functions in your Project for which you can specify data ranges. This Column displays three expandable menu items: • **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.

	XML File Entry:
	
	`<function name = "funcName" ...>`
	
	`<scalar name = "arg1" ...>`
	
	`<pointer name = "arg2" ...>`

File	Displays the name of the source file containing the variable or function.
	XML File Entry: `<file name = "C:\Project1\Sources\file.c" ...>`
Attributes	Displays information about the variable or function. For example, static variables display `static`. Uncalled functions display `unused`.

Column	Settings
	XML File Entry: `<function name="funcName" attributes="unused" ...>`
Data Type	Displays the variable type.
	XML File Entry: `<scalar name="arg1" complete_type="int32" ...>`
Main Generator Called	Applicable only for user-defined functions. Specifies whether the main generator calls the function: • **`MAIN GENERATOR`** – Main generator may call this function, depending on the value of the `-functions-called-in-loop` (C) or `-main-generator-calls` (C++) parameter. • **`NO`** – Main generator will not call this function. • **`YES`** – Main generator will call this function.
	XML File Entry: `<function name="funcName" main_generator_called="MAIN_GENERATOR" ...>`
Init Mode	Specifies how the software assigns a range to the variable: • **`MAIN GENERATOR`** – Variable range is assigned depending on the settings of the main generator options `-main-generator-writes-variables` and `-no-def-init-glob`. • **`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 `globalassert` mode if you need a warning. User-defined functions support only `INIT` mode. Stub functions support only `PERMANENT` mode. For C verifications, global pointers support `MAIN GENERATOR`, `IGNORE`, or `INIT` mode. • **`MAIN GENERATOR`** – Pointer follows the options of the main generator. • **`IGNORE`** – Pointer is not initialized • **`INIT`** – Specify if the pointer is NULL, and how the pointed object is allocated (**Initialize Pointer** and **Init Allocated** options).
	XML File Entry: `<scalar name="arg1" init_mode="INIT" ...>`

Column	Settings

Init Range	Specifies the minimum and maximum values for the variable.
	
	You can use the keywords `min` and `max` to denote the minimum and maximum values of the variable type. For example, for the type long, `min` and `max` correspond to -2^31 and 2^31-1 respectively.
	
	You can also use hexadecimal values. For example: `0x12..0x100`
	
	For `enum` variables, you cannot specify ranges directly using the enumerator constants. Instead use the values represented by the constants.
	
	For `enum` variables, you can also use the keywords `enum_min` and `enum_max` to denote the minimum and maximum values that the variable can take. For example, for an `enum` variable of the type defined below, `enum_min` is 0 and `enum_max` is 5:
	
	`enum week{ sunday, monday=0, tuesday,`
	`     wednesday, thursday, friday, saturday};`
	
	XML File Entry:
	
	`<scalar name="arg1" init_range="-1..0"...>`

Initialize Pointer	Applicable only to pointers. Enabled only when you specify **Init Mode**:`INIT`. Specifies whether the pointer should be NULL: • **`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. **Note** Not applicable for C++ projects. See “Constraint Specification Limitations”. **XML File Entry**: `<pointer name="arg1" initialize_pointer="Not NULL"...>`
Init Allocated	Applicable only to pointers. Enabled only when you specify **Init Mode**:`INIT`. Specifies how the pointed object is allocated: • **`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. **Note** Not applicable for C++ projects. See “Constraint Specification Limitations”.

Column	Settings
	XML File Entry: `<pointer name="arg1" init_pointed="MAIN_GENERATOR"...>`

# Allocated Objects	Applicable only to pointers. 	
	
	Specifies how many objects are pointed to by the pointer (the pointed object is considered as an array).
	
	The **Init Allocated** parameter specifies how many allocated objects are actually initialized. For instance, consider this code:
	
	`void func(int *ptr) {`
	`    assert(ptr[0]==1);`
	`    assert(ptr[1]==1);`
	`}`
	
	If you specify these constraints:
	
	• `ptr` has **Init Allocated** set to `MULTI` and **# Allocated Objects** set to 2,
	• `*ptr` has **Init Range** set to `1..1`,
	
	both assertions are green. However, if you specify these constraints:
	
	• `ptr` has **Init Allocated** set to `SINGLE`
	• `*ptr` has **Init Range** set to `1..1`,
	
	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.
	
	Use the keyword "max" to specify that a pointer can point to anywhere within a very large array of unknown file size. In your analysis results, you see very large offsets and buffer sizes for the pointer. The offset and buffer sizes depend on the pointer size and other characteristics of the target processor type used. The largest object Polyspace creates using this method is a buffer of 2^27-1 bytes (134217726 bytes).
	
	Note Not applicable for C++ projects. See “Constraint Specification Limitations”.
	
	XML File Entry:
	
	`<pointer name="arg1" number_allocated="10"...>`

Global Assert	Specifies whether to perform an assert check on the variable at global initialization, and after each assignment. **XML File Entry**: `<scalar name="glob" global_assert="YES"...>`
Global Assert Range	Specifies the minimum and maximum values for the range you want to check. **XML File Entry**: `<scalar name="glob" assert_range="0..200"...>`

<table>
<tr><th>Column</th><th>Settings</th></tr>
<tr><td rowspan="2">Comment</td><td>Remarks that you enter, for example, justification for your DRS values.</td></tr>
<tr><td>XML File Entry:
<code><scalar name="glob" comment="Speed Range"...></code></td></tr>
</table>

Constraint Specification Limitations

You cannot specify the following types of constraints using the constraint specification interface. To work around some of these limitations, you can define your own stubs and `main()`. For details, see "Constrain Variable Ranges for Polyspace Analysis Using Manual Stubs and Manual main() Function" on page 18-18.

The constraint specification interface does not support these kinds of constraints:

- In C++, you cannot constrain pointer or reference arguments of functions.

  Because of polymorphism, a C++ pointer or reference can point to objects of multiple classes in a class hierarchy and can require invoking different constructors. The pre-analysis for constraint specification cannot determine which object type to constrain or which constructor to call.
- You cannot specify a constraint that relates the return value of a function to its inputs. You can specify only a constant range for the constraints.
- You cannot specify multiple ranges for a constraint. For instance, you cannot specify that a function argument has either the value -1 or a value in the range [1,100]. Instead, specify the range [-1,100] or perform two separate analyses, once with the value -1 and once with the range [1,100].
- You cannot specify separate constraints on different fields of a union.

See Also

More About

- "Specify External Constraints for Polyspace Analysis" on page 14-2
- "Constrain Variable Ranges for Polyspace Analysis Using Manual Stubs and Manual main() Function" on page 18-18

Constrain Global Variable Range for Polyspace Analysis

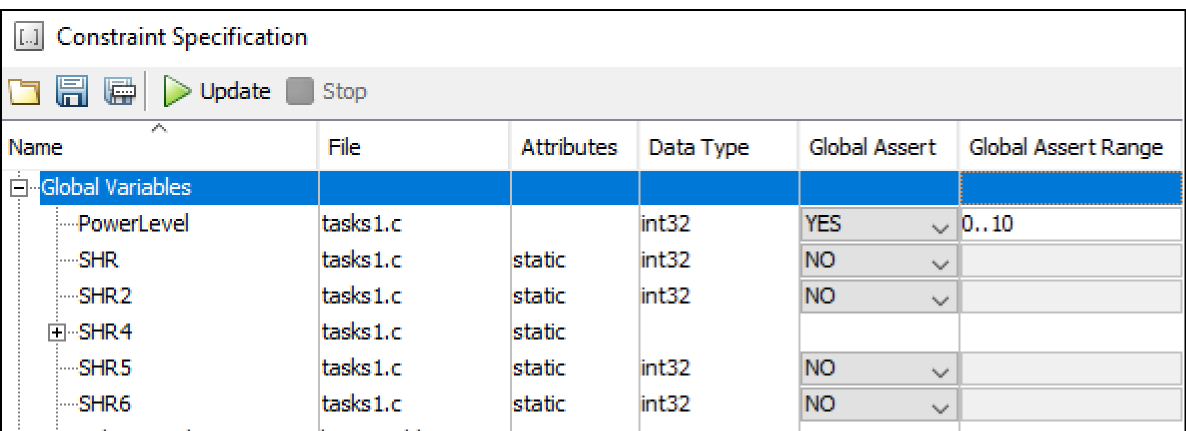
You can impose external constraints (also known as data range specifications or DRS) on the range of global variables in C/C++ code and check with Polyspace Code Prover whether write operations on the variable violate the constraint. For the general workflow, see “Specify External Constraints for Polyspace Analysis” on page 14-2.

User Interface (Desktop Products Only)

To constrain a global variable range and also check for violation of the constraint:

1. In your project configuration, select **Inputs & Stubbing**. Click the Edit button next to the **Constraint setup** field.
2. In the Constraint Specification window, click Generate.

   Under the **Global Variables** node, you see a list of global variables.



3. For the global variable that you want to constrain:

   - From the drop-down list in the **Global Assert** column, select `YES`.
   - In the **Global Assert Range** column, enter the range in the format `min..max`. `min` is the minimum value and `max` the maximum value for the global variable.
4. To save your specifications, click the button.

   In **Save a Constraint File** window, save your entries as an `xml` file.
5. Run a verification and open the results.

   For every write operation on the global variable, you see a green, orange, or red **Correctness condition** check. If the check is:

   - Green, the variable is within the range that you specified.
   - Orange, the variable can be outside the range that you specified.
   - Red, the variable is outside the range that you specified.

When two or more tasks write to the same global variable, the **Correctness condition** check can appear orange on all write operations to the variable even when only one write operation takes the variable outside the **Global Assert** range.

Command Line

Use the option `Constraint setup (-data-range-specifications)` with an XML file specifying your constraint.

For instance, for an analysis with Polyspace Code Prover Server, specify the option as follows:

```
polyspace-code-prover-server -sources filename
        -data-range-specifications "C:\Polyspace\drs_project1.xml"
```

Create a blank constraint XML template as described in “Specify External Constraints for Polyspace Analysis” on page 14-2. In the XML file, locate and constrain the global variables. XML tags for global variables appear directly within the `file` tag without an enclosing `function` tag. For instance, in this constraint XML, `PowerLevel` and `SHR` are global variables:

```
<file name="\\\\home\\Polyspace_Workspace\\Examples\\Code_Prover_Example
                                              \\sources\\tasks1.c">
   <scalar name="PowerLevel" line="26" .. global_assert="YES" assert_range="0..10"/>
   <scalar name="SHR" line="30" ... global_assert="NO" assert_range="" />
   <function name="Tserver" line="73" .../>
   <function name="initregulate" line="47" .../>
   <function name="orderregulate" line="35" ...>
     <scalar name="return" ... global_assert="unsupported" assert_range="unsupported" />
   </function>
   <function name="proc1" line="101" .../>
</file>
```

To specify a constraint on a global variable and check during a Code Prover analysis if the constraint is violated:

1. Set the `global_assert` attribute of the variable's `scalar` tag to `YES`.
2. Set the `assert_range` attribute to a range in the form *`min..max`*, for instance, `0..10`.

In the preceding example, the variable `PowerLevel` is constrained this way.

See Also

Polyspace Analysis Options
`Constraint setup (-data-range-specifications)`

Polyspace Results
`Correctness condition`

More About

- “Specify External Constraints for Polyspace Analysis” on page 14-2
- “External Constraints for Polyspace Analysis” on page 14-6
- “Constrain Function Inputs for Polyspace Analysis” on page 14-14

Constrain Function Inputs for Polyspace Analysis

If a program module analyzed with Polyspace Code Prover does not contain a `main` function, the analysis by default starts effectively from all uncalled functions[1]. Since these functions are not called within the code, Code Prover has to make assumptions about the function inputs based on their data types. For a more precise Code Prover analysis, you can specify constraints (also known as data range specifications or DRS) on these function inputs. Code Prover analyzes these functions for run-time errors with respect to the constrained inputs. For the general workflow, see "Specify External Constraints for Polyspace Analysis" on page 14-2.

For instance, for a function defined as follows, you can specify that the argument `val` has values in the range `[1..10]`. You can also specify that the argument `ptr` points to a 3-element array where each element is initialized:

```
int func(int val, int* ptr) {
      .
      .
}
```

A similar assumption about function inputs is seen in Bug Finder if you use the option `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`. You can also constrain a Bug Finder analysis with external constraints.

Note that if a function is called within the code, the external constraints no longer apply. Code Prover tracks the data flow within the code and analyzes a called function with actual arguments used in the code.

User Interface (Desktop Products Only)

To specify constraints on function inputs:

1. In your project configuration, select **Inputs & Stubbing**. Click the Edit button for **Constraint setup**.
2. In the Constraint Specification window, click Generate.

   Under the **User Defined Functions** node, you see a list of functions whose inputs can be constrained.
3. Expand the node for each function.

   You see each function input on a separate row. The inputs have the syntax *`function_name`*`.arg1`, *`function_name`*`.arg2`, etc.
4. Specify your constraints on one or more of the function inputs. For more information, see "External Constraints for Polyspace Analysis" on page 14-6.

   For example, in the preceding code:

   - To constrain `val` to the range `[1..10]`, select `INIT` for **Init Mode** and enter `1..10` for **Init Range**.

---

1 The Code Prover analysis generates a `main` that calls all uncalled functions by default and starts analysis from this `main`. You can change this default behavior using the option `Functions to call (-main-generator-calls)`.

- To specify that `ptr` points to a 3-element array where each element is initialized, select `MULTI` for **Init Allocated** and enter 3 for **# Allocated Objects**.

Constraint Specification

Update Stop Search

Name	File	Attributes	Data Type	Main Generator Called	Init Mode	Init Range	Initialize Pointer	Init Allocated	# Allocated Objects
Global Variables									
User Defined Functions									
func()	file.c	unused		MAIN GENERATOR					
func.arg1	file.c		int32		INIT	1..10			
func.arg2	file.c		int32 *		INIT		Not NULL	MULTI	3
func.* arg2	file.c		int32		MAIN GENE...				
func.return	file.c		int32						

5. Run verification and open the results. On the **Source** pane, place your cursor on the function inputs.

   The tooltips display the constraints. For example, in the preceding code, the tooltip displays that `val` has values in `1..10`.

Command Line

Use the option `Constraint setup (-data-range-specifications)` with an XML file specifying your constraint.

For instance, for an analysis with Polyspace Code Prover Server, specify the option as follows:

```
polyspace-code-prover-server -sources filename
          -data-range-specifications "C:\Polyspace\drs_project1.xml"
```

Create a blank constraint XML template as described in "Specify External Constraints for Polyspace Analysis" on page 14-2. In the XML file, locate and constrain the function inputs. The function inputs appear as a `scalar` or `pointer` tag in a `function` tag. The inputs are named as `arg1`, `arg2` and so on. For instance, for the preceding code, the XML structure for the inputs of `func` appear as follows:

```
<function name="func" line="1" attributes="unused"
    main_generator_called="MAIN_GENERATOR" comment="">
     <scalar name="arg1" line="1" base_type="int32"
            complete_type="int32" init_mode="INIT" init_range="1..10" />
     <pointer name="arg2" line="1" complete_type="int32 *"
            init_mode="INIT" initialize_pointer="Not NULL" number_allocated="3"
            init_pointed="MULTI">
         <scalar line="1" base_type="int32" complete_type="int32"
               init_mode="MAIN_GENERATOR" init_range=""/>
    </pointer>
     <scalar name="return" line="1" base_type="int32" complete_type="int32"
               init_mode="disabled" init_range="disabled"/>
</function>
```

To specify a constraint on a function input, set the attributes `init_mode` and `init_range` for scalar variables, and `init_pointed` and `number_allocated` for pointer variables.

- To constrain `val` to the range `[1..10]`, set the `init_mode` attribute of the tag with name `arg1` to `INIT` and `init_range` to `1..10`.

- To specify that `ptr` points to a 3-element array where each element is initialized, set the `init_mode` attribute of the tag with name `arg2` to `INIT`, `init_pointed` to `MULTI` and `number_allocated` to `3`.

See Also

`Constraint setup (-data-range-specifications)`

More About

- “Specify External Constraints for Polyspace Analysis” on page 14-2
- “External Constraints for Polyspace Analysis” on page 14-6
- “Constrain Global Variable Range for Polyspace Analysis” on page 14-12

XML File Format for Polyspace Analysis Constraints

For a more precise Polyspace analysis, you can specify constraints on global variables, function inputs and stubbed functions. You can specify the constraints in the user interface of the Polyspace desktop products or at the command line as an XML file. For the general workflow, see "Specify External Constraints for Polyspace Analysis" on page 14-2.

This topic describes details of the constraint XML file schema. You typically require this information only if you create a constraint XML from scratch. If you run a verification once, the software automatically generates a template constraint file `drs-template.xml` in your results folder. Instead of creating a constraint XML file from scratch, it is easier to edit this template XML file to specify your constraints. For some examples, see:

- "Constrain Global Variable Range for Polyspace Analysis" on page 14-12
- "Constrain Function Inputs for Polyspace Analysis" on page 14-14

For another explanation of what the XML tags mean, see "External Constraints for Polyspace Analysis" on page 14-6.

You can also see the information in this topic and the underlying XML schema in *`polyspaceroot`* `\polyspace\drs`. Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

Syntax Description — XML Elements

The constraints file contains the following XML elements:

- `<global>` element — Declares the global scope, and is the root element of the XML file.
- `<file>` element — Declares a file scope. Must be enclosed in the `<global>` element. May enclose any variable or function declaration. Static variables must be enclosed in a file element to avoid conflicts.
- `<scalar>` element— Declares an integer or a floating point variable. May be enclosed in any recognized element, but cannot enclose any element. Sets init/permanent/global asserts on variables.
- `<pointer>` element — Declares a pointer variable. May enclose any other variable declarations (including itself), to define the pointed objects. Specifies what value is written into pointer (NULL or not), how many objects are allocated and how the pointed objects are initialized.
- `<array>` element — Declares an array variable. May enclose any other variable definition (including itself), to define the members of the array.
- `<struct>` element — Declares a structure variable or object (instance of class). May enclose any other variable definition (including itself), to define the fields of the structure.
- `<function>` element — Declares a function or class method scope. May enclose any variable definition, to define the arguments and the return value of the function. Arguments should be named *`arg1, arg2, …argn`* and the return value should be called *`return`*.

The following notes apply to specific fields in each XML element:

- **(*)** — Fields used only by the GUI. These fields are not mandatory for verification to accept the ranges. The field line contains the line number where the variable is declared in the source code, `complete_type` contains a string with the complete variable type, and `base_type` is used by the

GUI to compute the min and max values. The field comment is used to add information about any node.

- **(**)** — The field name is mandatory for scope elements `<file>` and `<function>` (except for function pointers). For other elements, the name must be specified when declaring a root symbol or a `struct` field.
- **(***)** — If more than one attribute applies to the variable, the attributes must be separated by a space. Only the static attribute is mandatory, to avoid conflicts between static variables having the same name. An attribute can be defined multiple times without impact.
- **(****)** — This element is used only by the GUI, to determine which `init` modes are allowed for the current element (according to its type). The value works as a mask, where the following values are added to specify which modes are allowed:

  - **1**: The mode "`NO`" is allowed.
  - **2** : The mode "`INIT`" is allowed.
  - **4**: The mode "`PERMANENT`" is allowed.
  - **8**: The mode "`MAIN_GENERATOR`" is allowed.

  For example, the value "**10**" means that modes "`INIT`" and "`MAIN_GENERATOR`" are allowed. To see how this value is computed, refer to "Valid Modes and Default Values".
- **(*****)** — 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`	*`filepath_or_filename`*
`comment`	*`string`*

<scalar> Element

Field	Syntax
`name` (**)	*`name`*
`line` (*)	*`line`*

`base_type` (*)	`intx`
	`uintx`
	`floatx`

`Attributes` (***)	`volatile`
	`extern`
	`static`
	`const`

`complete_type` (*)	*`type`*

Field	Syntax
init_mode	MAIN_GENERATOR IGNORE INIT PERMANENT disabled unsupported
init_modes_allowed (*)	*single value* (****)
init_range	*range* disabled unsupported
global_ assert	YES NO disabled unsupported
assert_range	*range* disabled unsupported
comment(*)	*string*

<pointer> Element

Field	Syntax
Name (**)	*name*
line (*)	*line*
Attributes (***)	volatile extern static const
complete_type (*)	*type*
init_mode	MAIN_GENERATOR IGNORE INIT PERMANENT disabled unsupported
init_modes_allowed (*)	*single value* (****)
initialize_ pointer	May be: NULL Not NULL NULL
number_ allocated	*single value* disabled unsupported

Field	Syntax
init_pointed (******)	MAIN_GENERATOR NONE SINGLE MULTI SINGLE_CERTAIN_WRITE MULTI_CERTAIN_WRITE disabled
comment	*string*

<array> and <struct> Elements

Field	Syntax
Name (**)	*name*
line (*)	*line*
complete_type (*)	*type*
attributes (***)	volatile extern static const
comment	*string*

<function> Element

Field	Syntax
Name (**)	*name*
line (*)	*line*
main_generator_called	MAIN_GENERATOR YES NO disabled
attributes (***)	static extern unused
comment	*string*

Valid Modes and Default Values

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
Global variables	Base type	Unqualified/ static/ const scalar	MAIN_ GENERATOR IGNORE INIT	YES NO			Main generator dependent
			PERMANENT	disabled			
		Volatile scalar	PERMANENT	disabled			PERMANENT min..max
		Extern scalar	INIT	YES NO			INIT min..max
			PERMANENT	disabled			
	Struct	Struct field	Refer to field type				
	Array	Array element	Refer to element type				
Global variables	Pointer	Unqualified/ static/ const scalar	MAIN_ GENERATOR IGNORE INIT		May be NULL Not NULL NULL	NONE SINGLE MULTI	Main generator dependent
		Volatile pointer	un- supported		un- supported	un- supported	
		Extern pointer	IGNORE INIT		May be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May be NULL max MULTI
		Pointed volatile scalar	un- supported	un- supported			
		Pointed extern scalar	INIT	un- supported			INIT min..max
		Pointed other scalars	MAIN_ GENERATOR INIT	un- supported			MAIN_ GENERATOR dependent
		Pointed pointer	MAIN_ GENERATOR INIT/	un- supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	MAIN_ GENERATOR dependent
		Pointed function	un- supported	un- supported			
Function parameters	Userdef function	Scalar parameters	MAIN_ GENERATOR INIT	un- supported			INIT min..max

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
		Pointer parameters	MAIN_ GENERATOR INIT	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May be NULL max MULTI
		Other parameters	Refer to parameter type				
	Stubbed function	Scalar parameter	disabled	un-supported			
		Pointer parameters	disabled		disabled	NONE SINGLE MULTI SINGLE_ CERTAIN_ WRITE MULTI_ CERTAIN_ WRITE	MULTI
		Pointed parameters	PERMANENT	un-supported			PERMANENT min..max
		Pointed const parameters	disabled	un-supported			
Function return	Userdef function	Return	disabled	un-supported	disabled	disabled	
	Stubbed function	Scalar return	PERMANENT	un-supported			PERMANENT min..max
		Pointer return	PERMANENT	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI SINGLE_ CERTAIN_ WRITE MULTI_ CERTAIN_ WRITE	PERMANENT May be NULL max MULTI

See Also

More About

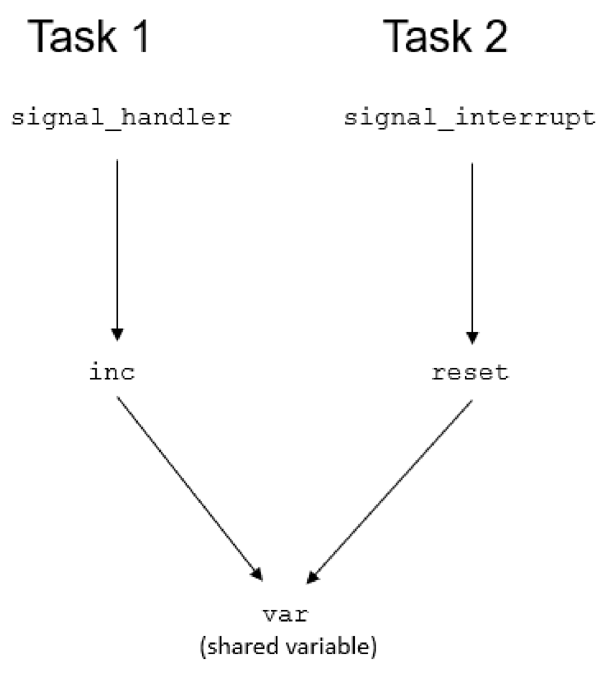
- "Specify External Constraints for Polyspace Analysis" on page 14-2
- "Constrain Global Variable Range for Polyspace Analysis" on page 14-12
- "Constrain Function Inputs for Polyspace Analysis" on page 14-14

Configure Multitasking Analysis

Analyze Multitasking Programs in Polyspace

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

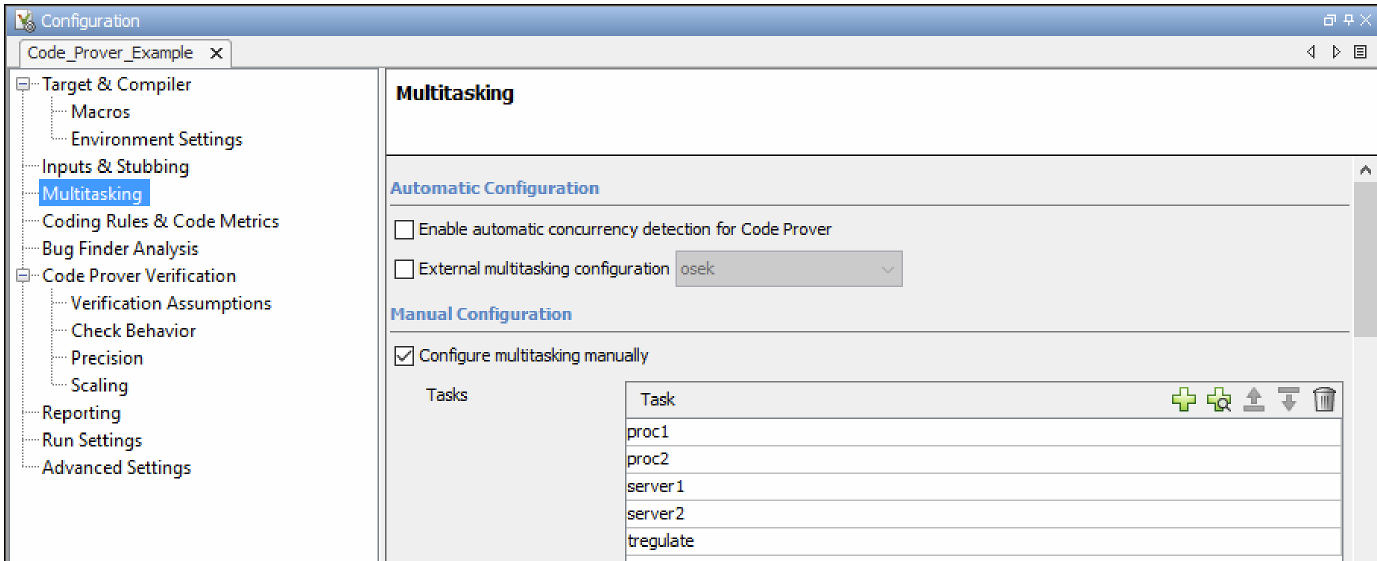
```
int var;

void reset(void) {
  var=0;
}

void inc(void) {
   var+=2;
}

/* Task 1 */
void signal_handler(void) {
  volatile int randomValue = 0;
  while(randomValue) {
     inc();
    }
}

/* Task 2 */
void signal_interrupt(void) {
  volatile int randomValue = 0;
  while(randomValue) {
     reset();
    }
}

void main() {
}
```



In addition to regular run-time checks, the analysis looks for issues specific to concurrent execution:

- Data races, deadlocks, consecutive or missing locks and unlocks (Bug Finder)
- Unprotected shared variables (Code Prover)

Configure Analysis



If your code uses multitasking primitives from certain families, for instance, `pthread_create` for thread creation:

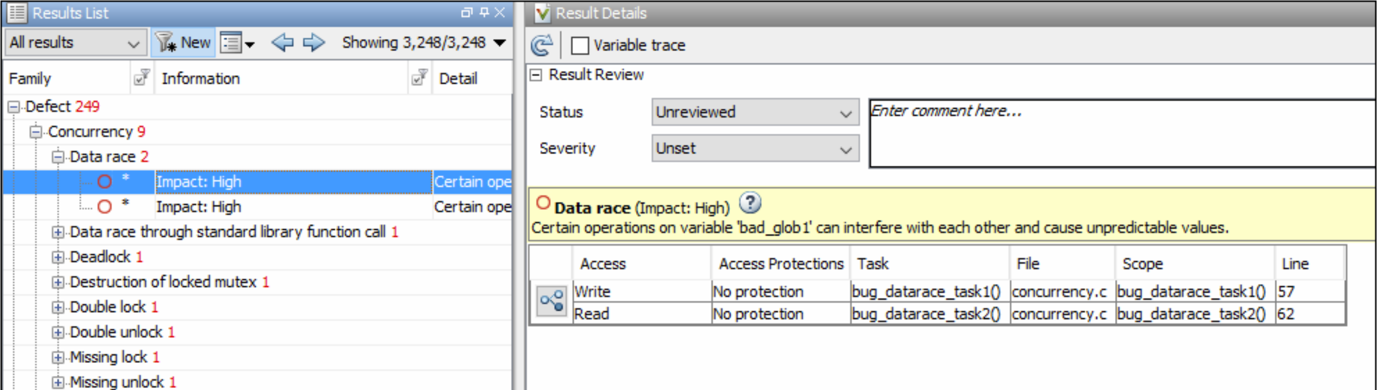
- In Bug Finder, the analysis detects them and extracts your multitasking model from the code.
- In Code Prover, you must enable this automatic detection explicitly. See `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.

See “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 15-7.

Alternatively, define your multitasking model through the analysis options. In the user interface of the Polyspace desktop products, the options are on the **Multitasking** node in the **Configuration** pane. Most options are common between Bug Finder and Code Prover. The multitasking analysis in Code Prover is more exhaustive about finding potentially unprotected shared variables and therefore follows a stricter model. Your code must be written in a specific format for Code Prover to successfully complete a multitasking analysis. For instance, the functions that you specify as entry points must be `void(void)` functions. However, if your code is not already written in this format, you can work around the restrictions. For details, see “Configuring Polyspace Multitasking Analysis Manually” on page 15-17.

Review Analysis Results

Bug Finder

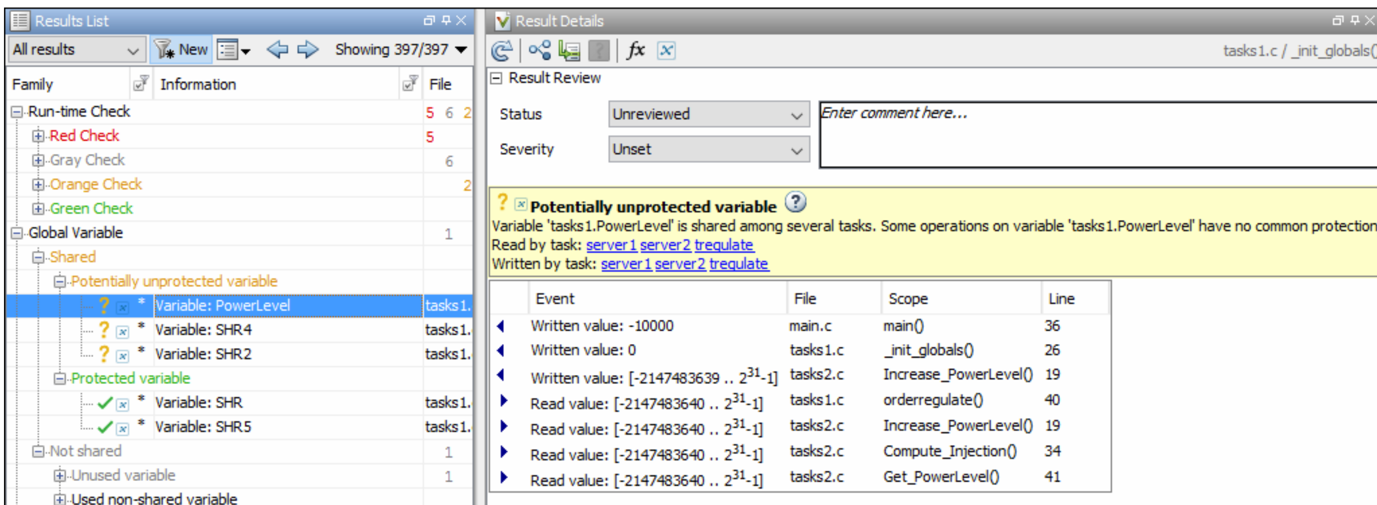


A Bug Finder analysis can find many different kinds of concurrency defects including:

- Data races, when operations on a variable from different tasks interfere with each other.
- Deadlocks or double locks, because of incorrect placement of lock and unlock functions

For the complete list, see “Concurrency Defects”. However, the analysis makes certain assumptions to avoid false positives, and might not find all data races. You can perform an initial check for data races with Bug Finder, and make a more exhaustive pass later with Code Prover.

Code Prover



The Code Prover analysis exhaustively checks if shared global variables are protected from concurrent access. The analysis reports variables that are definitely protected in green and variables that might be unprotected in orange. See “Global Variables”.

Review the results using the message on the **Result Details** pane. See a visual representation of conflicting operations using the (graph) icon.

Differences Between Bug Finder and Code Prover

The following table summarizes the differences between the multitasking analysis in Polyspace Bug Finder and Polyspace Code Prover.

Configuration

	Bug Finder	Code Prover
Auto-detection of concurrency routines	Supported by default	Supported on option `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`
Constraints on `main` function	None	The `main` function must terminate. It cannot contain an infinite loop or a run-time error. For workarounds if there is an intentional infinite loop in `main`, see "Adapt Code for Code Prover Multitasking Analysis" on page 15-17.
Atomic operations	Depending on the target size, certain operations are considered as atomic (non-interruptable). To consider all operations as non-atomic, use the option `-detect-atomic-data-race`. See also `Define Atomic Operations in Multitasking Code`.	All operations are considered as non-atomic.

Results

	Bug Finder	Code Prover
Concurrent unprotected access on shared variables (data races)	Shown using one of these results: • `Data race` • `Data race on adjacent bit fields` • `Data race through standard library function call`	Shown using the result `Potentially unprotected variable`. Code Prover is more exhaustive when keeping track of control and data flows. Therefore, Code Prover might detect probable data races not detected with Bug Finder.
Issues with concurrency routines besides data race: • Deadlocks, double locks, missing unlocks, and so on. • Improper thread creation, joining or destruction. • Memory escape from threads	Detected	Not detected

See Also

More About

- "Auto-Detection of Thread Creation and Critical Section in Polyspace" on page 15-7
- "Configuring Polyspace Multitasking Analysis Manually" on page 15-17
- "Protections for Shared Variables in Multitasking Code" on page 15-21

Auto-Detection of Thread Creation and Critical Section in Polyspace

With Polyspace, you can analyze programs where multiple threads run concurrently. Polyspace can analyze your multitasking code for data races, deadlocks and other concurrency defects, if the analysis is aware of the concurrency model in your code. In some situations, Polyspace can detect thread creation and critical sections in your code automatically. Bug Finder detects them by default. In Code Prover, you enable automatic detection using the option `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.

For the multitasking code analysis workflow, see “Analyze Multitasking Programs in Polyspace” on page 15-2.

If your thread creation function is not detected automatically:

- You can also map the function to a thread-creation function that Polyspace can detect automatically. Use the option `-code-behavior-specifications`.
- Otherwise, you must manually model your multitasking threads by using configuration options. See “Configuring Polyspace Multitasking Analysis Manually” on page 15-17.

Multitasking Routines that Polyspace Can Detect

Polyspace can detect thread creation and critical sections if you use primitives from these groups. Polyspace recognizes calls to these routines as the creation of a new thread or as the beginning or end of a critical section.

POSIX

Thread creation: `pthread_create`

Critical section begins: `pthread_mutex_lock`

Critical section ends: `pthread_mutex_unlock`

VxWorks

Thread creation: `taskSpawn`

Critical section begins: `semTake`

Critical section ends: `semGive`

To activate automatic detection of concurrency primitives for VxWorks®, in the user interface of the Polyspace desktop products, use the `VxWorks` template. For more information on templates, see “Create Project in Polyspace Desktop User Interface Using Configuration Template” on page 2-13. At the command-line, use these options:

```
-D1=CPU=I80386
-D2=__GNUC__=2
-D3=__OS_VXWORKS
```

Concurrency detection is possible only if the multitasking functions are created from an entry point named `main`. If the entry point has a different name, such as `vxworks_entry_point`, do one of the following:

- Provide a `main` function.
- `Preprocessor definitions (-D)`: In preprocessor definitions, set `vxworks_entry_point=main`.

Windows

Thread creation: `CreateThread`

Critical section begins: `EnterCriticalSection`

Critical section ends: `LeaveCriticalSection`

µC/OS II

Thread creation: `OSTaskCreate`

Critical section begins: `OSMutexPend`

Critical section ends: `OSMutexPost`

C++11

Thread creation: `std::thread::thread`

Critical section begins: `std::mutex::lock`

Critical section ends: `std::mutex::unlock`

For autodetection of C++11 threads, explicitly specify paths to your compiler header files or use `polyspace-configure`.

For instance, if you use `std::thread` for thread creation, explicitly specify the path to the folder containing `thread.h`.

See also “Limitations of Automatic Thread Detection”.

C11

Thread creation: `thrd_create`

Critical section begins: `mtx_lock`

Critical section ends: `mtx_unlock`

Example of Automatic Thread Detection

The following multitasking code models five philosophers sharing five forks. The example uses POSIX® thread creation routines and illustrates a classic example of a deadlock. Run Bug Finder on this code to see the deadlock.

```
#include "pthread.h"
#include <stdio.h>
#include <unistd.h>

pthread_mutex_t forks[5];

void* philo1(void* args)
{
    while (1) {
        printf("Philosopher 1 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[0]);
        printf("Philosopher 1 takes left fork\n");
        pthread_mutex_lock(&forks[1]);
        printf("Philosopher 1 takes right fork\n");
        printf("Philosopher 1 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[1]);
        printf("Philosopher 1 puts down right fork\n");
        pthread_mutex_unlock(&forks[0]);
        printf("Philosopher 1 puts down left fork\n");
    }
    return NULL;
}

void* philo2(void* args)
{
    while (1) {
        printf("Philosopher 2 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[1]);
        printf("Philosopher 2 takes left fork\n");
        pthread_mutex_lock(&forks[2]);
        printf("Philosopher 2 takes right fork\n");
        printf("Philosopher 2 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[2]);
        printf("Philosopher 2 puts down right fork\n");
        pthread_mutex_unlock(&forks[1]);
        printf("Philosopher 2 puts down left fork\n");
    }
    return NULL;
}

void* philo3(void* args)
{
    while (1) {
        printf("Philosopher 3 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[2]);
        printf("Philosopher 3 takes left fork\n");
        pthread_mutex_lock(&forks[3]);
        printf("Philosopher 3 takes right fork\n");
        printf("Philosopher 3 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[3]);
        printf("Philosopher 3 puts down right fork\n");
```

```
        pthread_mutex_unlock(&forks[2]);
        printf("Philosopher 3 puts down left fork\n");
    }
    return NULL;
}

void* philo4(void* args)
{
    while (1) {
        printf("Philosopher 4 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[3]);
        printf("Philosopher 4 takes left fork\n");
        pthread_mutex_lock(&forks[4]);
        printf("Philosopher 4 takes right fork\n");
        printf("Philosopher 4 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[4]);
        printf("Philosopher 4 puts down right fork\n");
        pthread_mutex_unlock(&forks[3]);
        printf("Philosopher 4 puts down left fork\n");
    }
    return NULL;
}

void* philo5(void* args)
{
    while (1) {
        printf("Philosopher 5 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[4]);
        printf("Philosopher 5 takes left fork\n");
        pthread_mutex_lock(&forks[0]);
        printf("Philosopher 5 takes right fork\n");
        printf("Philosopher 5 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[0]);
        printf("Philosopher 5 puts down right fork\n");
        pthread_mutex_unlock(&forks[4]);
        printf("Philosopher 5 puts down left fork\n");
    }
    return NULL;
}

int main(void)
{
    pthread_t ph[5];
    pthread_create(&ph[0], NULL, philo1, NULL);
    pthread_create(&ph[1], NULL, philo2, NULL);
    pthread_create(&ph[2], NULL, philo3, NULL);
    pthread_create(&ph[3], NULL, philo4, NULL);
    pthread_create(&ph[4], NULL, philo5, NULL);

    pthread_join(ph[0], NULL);
    pthread_join(ph[1], NULL);
    pthread_join(ph[2], NULL);
    pthread_join(ph[3], NULL);
    pthread_join(ph[4], NULL);
```

```
    return 1;
}
```

Each philosopher needs two forks to eat, a right and a left fork. The functions `philo1`, `philo2`, `philo3`, `philo4`, and `philo5` represent the philosophers. Each function requires two `pthread_mutex_t` resources, representing the two forks required to eat. All five functions run at the same time in five concurrent threads.

However, a deadlock occurs in this example. When each philosopher picks up their first fork (each thread locks one `pthread_mutex_t` resource), all the forks are being used. So, the philosophers (threads) wait for their second fork (second `pthread_mutex_t` resource) to become available. However, all the forks (resources) are being held by the waiting philosophers (threads), causing a deadlock.

Naming Convention for Automatically Detected Threads

If you use a function such as `pthread_create()` to create new threads (tasks), each thread is associated with an unique identifier. For instance, in this example, two threads are created with identifiers `id1` and `id2`.

```
pthread_t* id1, id2;

void main()
{
    pthread_create(id1, NULL, start_routine, NULL);
    pthread_create(id2, NULL, start_routine, NULL);
}
```

If a data race occurs between the threads, the analysis can detect it. When displaying the results, the threads are indicated as `task_`*`id`*, where `id` is the identifier associated with the thread. In the preceding example, the threads are identified as `task_id1` and `task_id2`.

If a thread identifier is:

- Local to a function, the thread name shows the function.

  For instance, the thread created below appears as `task_f:id`

  ```
  void f(void)
  {
      pthread_t* id;
      pthread_create(id, NULL, start_routine, NULL);
  }
  ```
- A field of a structure, the thread name shows the structure.

  For instance, the thread created below appears as `task_a#id`

  ```
  struct {pthread_t* id; int x;} a;
  pthread_create(a.id,NULL,start_routine,NULL);
  ```
- An array member, the thread name shows the array.

  For instance, the thread created below appears as `task_tab[1]`.

```
pthread_t* tab[10];
pthread_create(tab[1],NULL,start_routine,NULL);
```

If you create two threads with distinct thread identifiers, but you use the same local variable name for the thread identifiers, the name of the second thread is modified to distinguish it from the first thread. For instance, the threads below appear as `task_func:id` and `task_func:id:1`.

```
void func()
{
    {
        pthread_t id;
        pthread_create(&id, NULL, &task, NULL);

    }
    {
        pthread_t id;
        pthread_create(&id, NULL, &task, NULL);

    }
}
```

Limitations of Automatic Thread Detection

The multitasking model extracted by Polyspace does not include some features. Polyspace cannot model:

- Thread priorities and attributes — Ignored by Polyspace.
- Recursive semaphores.
- Unbounded thread identifiers, such as `extern pthread_t ids[]` — Warning.
- Calls to concurrency primitive through high-order calls — Warning.
- Aliases on thread identifiers — Polyspace over-approximates when the alias is used.
- Termination of threads — Polyspace ignores `pthread_join` and `thrd_join`. Polyspace replaces `pthread_exit` and `thrd_exit` by a standard exit.
- (Polyspace Bug Finder only) Creation of multiple threads through multiple calls to the same function with different pointer arguments.

  Example

  In this example, Polyspace considers that only one thread is created.

  ```
  pthread_t id1, id2;
  void start(pthread_t* id)
  {
      pthread_create(id, NULL, start_routine, NULL);
  }
  void main()
  {
      start(&id1);
      start(&id2);
  }
  ```
- (Polyspace Code Prover only) Shared local variables — Only global variables are considered shared. If a local variable is accessed by multiple threads, the analysis does not take into account the shared nature of the variable.

Example

In this example, the analysis does not take into account that the local variable `x` can be accessed by both `task1` and `task2` (after the new thread is created).

```
#include <pthread.h>
#include <stdlib.h>


void* task2(void* args)
{
    int* x = (int*) args;
    *x = 1;
    return (void*)x;
}

void task1()
{
    int x;
    x = 2;
    pthread_t id;
    (void)pthread_create(&id, NULL, task2, (void*) &x);
    /* x (local var) passed to task2 */
    x = 3 ;

    /* Unknown thread priority means x = 1 OR x = 3.*/
    /* However, the analysis considers  x = 3 */
    /* Assertion below is green */
    assert(x == 3);
}

int main(void)
{
    task1();
    return 0;
}
```

- (Polyspace Code Prover only) Shared dynamic memory — Only global variables are considered shared. If a dynamically allocated memory region is accessed by multiple threads, the analysis does not take into account its shared nature.

Example

In this example, the analysis does not take into account that `lx` points to a shared memory region. The region can be accessed by both `task1` and `task2` (after the new thread is created). The Code Prover analysis also reports `lx` as a non-shared variable.

```
#include <pthread.h>
#include <stdlib.h>

static int* lx;

void* task2(void* args)
{
    int* x = (int*) args;
    *x = 1;
    return (void*)x;
}

void task1()
{
    pthread_t id;
    lx = (int*)malloc(sizeof(int));

    if (lx == NULL) exit(1);

    (void)pthread_create(&id, NULL, task2, (void*) lx);

    *lx = 3 ;

    /* Unknown thread priority means *lx = 1 OR *lx = 3.*/
    /* However, the analysis considers  *lx = 3 */
    /* Assertion below is green */
    assert(*lx == 3);
}

int main(void)
{
    task1();
    return 0;
}
```

- Number of tasks created with `CreateThread` when `threadId` is set to `NULL`— When you create multiple threads that execute the same function, if the last argument of `CreateThread` is `NULL`, Polyspace only detects one instance of this function, or task.

  Example

  In this example, Polyspace detects only one instance of `thread_function1()`, but 10 instances of `thread_function2()`.

```
#include <windows.h>

#define MAX_LOOP_THREADS 10

DWORD WINAPI thread_function1(LPVOID data) {}
DWORD WINAPI thread_function2(LPVOID data) {}

HANDLE hds1[MAX_LOOP_THREADS];
HANDLE hds2[MAX_LOOP_THREADS];
DWORD threadId[MAX_LOOP_THREADS];

int main(void)
{
    for (int i = 0; i < MAX_LOOP_THREADS; i++) {

      hds1[i] = CreateThread(NULL, 0, thread_function1, NULL, 0, NULL);
      hds2[i] = CreateThread(NULL, 0, thread_function2, NULL, 0, &threadId[i]);
    }

    return 0;
}
```

- (C++11 only) If you use lambda expressions as start functions during thread creation, Polyspace does not detect shared variables in the lambda expressions.

  Example

  In this example, Polyspace does not detect that the variable `y` used in the lambda expressions is shared between two threads. As a result, Bug Finder, for instance, does not show a **Data race** defect.

```
#include <thread>
int y;
int main() {
   std::thread t1([] {y++;});
   std::thread t2([] {y++;});
   t1.join();
   t2.join();
   return 0;
}
```

- (C++11 threads with Polyspace Code Prover only) String literals as thread function argument — Code Prover shows a red **Illegally dereferenced pointer** error if the thread function has an `std::string&` parameter and you pass a string literal argument.

  Example

  In this example, the thread function `foo` has an `std::string&` parameter. When starting a thread, a string literal is passed as argument to this function, which undergoes an implicit conversion to `std::string` type. Code Prover loses track of the original string literal in this conversion. Therefore, a dashed red underline appears on `operator<<` in the body of `foo` and a red **Illegally dereferenced pointer** check in the body of `operator<<`.

```
#include <iostream>
#include <thread>

using namespace std;

void foo(const std::string& f) {
    std::cout << f;
}

void main() {
  std::thread t1(foo,"foo_arg");
}
```

To work around this issue, assign the string literal to a temporary variable and pass the variable as argument to the thread function.

```
#include <iostream>
#include <thread>

using namespace std;

void foo(const std::string& f) {
    std::cout << f;
}

void main() {
  std::string str = "foo_arg";
  std::thread t1(foo, str);
}
```

See Also

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

More About

- “Analyze Multitasking Programs in Polyspace” on page 15-2
- “Configuring Polyspace Multitasking Analysis Manually” on page 15-17

Configuring Polyspace Multitasking Analysis Manually

With Polyspace, you can analyze programs where multiple threads run concurrently. In some situations, Polyspace can detect thread creation and critical sections in your code automatically. See “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 15-7.

If your code has functions that are intended for concurrent execution, but that cannot be detected automatically, you must specify them before analysis. If these functions operate on a common variable, you must also specify protection mechanisms for those operations.

For the multitasking code analysis workflow, see “Analyze Multitasking Programs in Polyspace” on page 15-2.

Specify Options for Multitasking Analysis

Use these options to specify cyclic tasks, interrupts and protections for shared variables. In the user interface of the Polyspace desktop products, the options are on the **Multitasking** node in the **Configuration** pane. The following options can be used in both a Bug Finder and Code Prover analysis:

- `Tasks (-entry-points)`: Specify noncyclic entry point functions.

  Do not specify `main`. Polyspace implicitly considers `main` as an entry point function.
- `Critical section details (-critical-section-begin -critical-section-end)`: Specify functions that begin and end critical sections.
- `Temporally exclusive tasks (-temporal-exclusions-file)`: Specify groups of functions that are temporally exclusive.

A Polyspace analysis supports four levels of task priorities. That is, the analysis can take into consideration the fact that certain tasks cannot be interrupted by tasks with lower priorities. You can use these options to indicate task priorities:

- `Cyclic tasks (-cyclic-tasks)`: Specify functions that are scheduled at periodic intervals.
- `Interrupts (-interrupts)`: Specify functions that can run asynchronously.
- `-preemptable-interrupts`: Specify functions that have lower priority than interrupts, but higher priority than tasks (preemptable or non-preemptable).
- `-non-preemptable-tasks`: Specify functions that have higher priority than tasks, but lower priority than interrupts (preemptable or non-preemptable).
- `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`: Specify functions that disable and reenable interrupts.

For an example of using priorities, see “Protections for Shared Variables in Multitasking Code” on page 15-21.

Adapt Code for Code Prover Multitasking Analysis

The multitasking analysis in Code Prover is more exhaustive about finding potentially unprotected shared variables and therefore follows a strict model.

Tasks and interrupts must be `void(void)` functions.

Functions that you specify as tasks and interrupts must have the prototype:

```
void func(void);
```

Suppose you want to specify a function `func` that takes `int` arguments and has return type `int`:

```
int func(int);
```

Define a wrapper void-void function that calls `func` with a volatile value. Specify this wrapper function as a task or interrupt.

```
void func_wrapper() {
  volatile int arg;
  (void)func(arg);
}
```

You can save the wrapper function definition along with a declaration of the original function in a separate file and add this file to the analysis.

The `main` function must end.

Code Prover assumes that the `main` function ends before all tasks and interrupts begin. If the `main` function contains an infinite loop or run-time error, the tasks and interrupts are not analyzed. If you see that there are no checks in your tasks and interrupts, look for a token underlined in dashed red to identify the issue in the `main` function. See "Reasons for Unchecked Code" on page 34-77.

Suppose you want to specify the `main` function as a cyclic task.

```
void performTask1Cycle(void);
void performTask2Cycle(void);

void main() {
 while(1) {
    performTask1Cycle();
  }
}

void task2() {
 while(1) {
    performTask2Cycle();
  }
}
```

Replace the definition of `main` with:

```
#ifdef POLYSPACE
void main() {
}
void task1() {
 while(1) {
    performTask1Cycle();
  }
}

#else
void main() {
 while(1) {
    performTask1Cycle();
```

```
  }
}
#endif
```

The replacement defines an empty `main` and places the content of `main` into another function `task1` if a macro `POLYSPACE` is defined. Define the macro `POLYSPACE` using the option `Preprocessor definitions (-D)` and specify `task1` for the option `Tasks (-entry-points)`.

This assumption does not apply to automatically detected threads. For instance, a `main` function can create threads using `pthread_create`.

The Polyspace multitasking analysis assumes that a task or interrupt cannot interrupt itself.

All tasks and interrupts can run any number of times in any sequence.

The Code Prover analysis considers that all tasks and interrupts can run any number of times in any sequence.

Suppose in this example, you specify `reset` and `inc` as cyclic tasks. The analysis shows an overflow on the operation `var+=2`.

```
void reset(void) {
 var=0;
}

void inc(void) {
  var+=2;
}
```

Suppose you want to model a scheduling of tasks such that `reset` executes after `inc` has executed five times. Write a wrapper function that implements this sequence. Specify this new function as a cyclic task instead of `reset` and `inc`.

```
void task() {
 volatile int randomValue = 0;
 while(randomValue) {
   inc();
   inc();
   inc();
   inc();
   inc();
   reset();
   }
 }
```

Suppose you want to model a scheduling of tasks such that `reset` executes after `inc` has executed zero to five times. Write a wrapper function that implements this sequence. Specify this new function as a cyclic task instead of `reset` and `inc`.

```
void task() {
 volatile int randomValue = 0;
 while(randomValue) {
   if(randomValue)
     inc();
   if(randomValue)
     inc();
   if(randomValue)
```

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

See Also

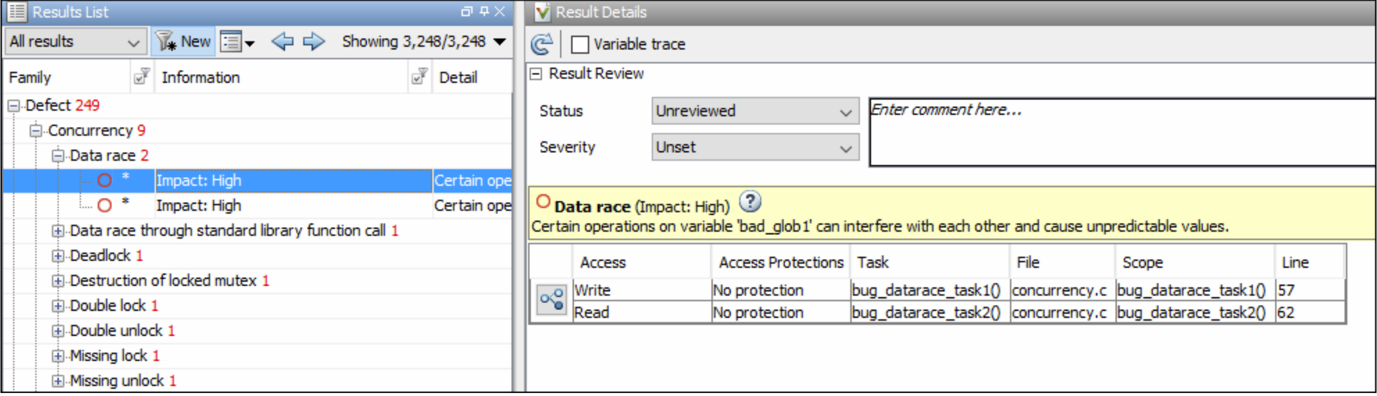
More About

- “Analyze Multitasking Programs in Polyspace” on page 15-2
- “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 15-7

Protections for Shared Variables in Multitasking Code

If your code is intended for multitasking, tasks in your code can access a common shared variable. To prevent data races, you can protect read and write operations on the variable. This topic shows the various protection mechanisms that Polyspace can recognize.

Detect Unprotected Access



You can detect an unprotected access using either Bug Finder or Code Prover. Code Prover is more exhaustive and proves if a shared variable is protected from concurrent access.

- Bug Finder detects an unprotected access using the result **Data race**. See `Data race`.
- Code Prover detects an unprotected access using the result **Shared unprotected global variable**. See `Potentially unprotected variable`.

Suppose you analyze this code, specifying `signal_handler_1` and `signal_handler_2` as cyclic tasks. Use the analysis option `Cyclic tasks (-cyclic-tasks)`.

```
#include <limits.h>
int shared_var;

void inc() {
     shared_var+=2;
}

void reset() {
     shared_var = 0;
}

void signal_handler_1(void) {
    reset();
    inc();
    inc();
}

void signal_handler_2(void) {
    shared_var = INT_MAX;
}

 void main() {
}
```

Bug Finder shows a data race on `shared_var`. Code Prover shows that `shared_var` is a potentially unprotected shared variable. Code Prover also shows that the operation `shared_var += 2` can overflow. The overflow occurs if the call to `inc` in `signal_handler_1` immediately follows the operation `shared_var = INT_MAX` in `signal_handler_2`.

Protect Using Critical Sections

One possible solution is to protect operations on shared variables using critical sections.

In the preceding example, modify your code so that operations on `shared_var` are in the same critical section. Use the functions `take_semaphore` and `give_semaphore` to begin and end the critical sections. To specify these functions that begin and end critical sections, use the analysis options `Critical section details (-critical-section-begin -critical-section-end)`.

```
#include <limits.h>
int shared_var;

void inc() {
 shared_var+=2;
}

void reset() {
 shared_var = 0;
}

/* Declare lock and unlock functions */
void take_semaphore(void);
void give_semaphore(void);

void signal_handler_1() {
    /* Begin critical section */
    take_semaphore();
    reset();
    inc();
    inc();
    /* End critical section */
    give_semaphore();

}

void signal_handler_2() {
   /* Begin critical section */
   take_semaphore();
   shared_var = INT_MAX;
   /* End critical section */
   give_semaphore();

}

void main() {
}
```

You do not see the data race in Bug Finder. Code Prover proves that the shared variable is protected. You also do not see the overflow because the call to `reset()` in `signal_handler_1` always precedes calls to `inc()`.

You can also use primitives such as the POSIX functions `pthread_mutex_lock` and `pthread_mutex_unlock` to begin and end critical sections. For a list of primitives that Polyspace can detect automatically, see "Auto-Detection of Thread Creation and Critical Section in Polyspace" on page 15-7.

Protect Using Temporally Exclusive Tasks

Another possible solution is to specify a group of tasks as temporally exclusive. Temporally exclusive tasks cannot interrupt each other.

In the preceding example, specify that `signal_handler_1` and `signal_handler_2` are temporally exclusive. Use the option `Temporally exclusive tasks (-temporal-exclusions-file)`.

You do not see the data race in Bug Finder. Code Prover proves that the shared variable is protected. You also do not see the overflow because the call to `reset()` in `signal_handler_1` always precedes calls to `inc()`.

Protect Using Priorities

Another possible solution is to specify that one task has higher priority over another.

In the preceding example, specify that `signal_handler_1` is an interrupt. Retain `signal_handler_2` as a cyclic task. Use the options `Cyclic tasks (-cyclic-tasks)` and `Interrupts (-interrupts)`.

Bug Finder does not show the data race defect anymore. The reason is this:

- The operation `shared_var = INT_MAX` in `signal_handler_2` is atomic. Therefore, the operations in `signal_handler_1` cannot interrupt it.
- The operations in `signal_handler_1` cannot be interrupted by the operation in `signal_handler_2` because `signal_handler_1` has higher priority.

You can specify up to four different priorities with these options (with highest priority listed first):

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

A task with higher priority is atomic with respect to a task with lower priority. Note that if you use the option `-detect-atomic-data-race`, the analysis ignores the difference in priorities and continues to show the data race. See also "Define Task Priorities for Data Race Detection in Polyspace" on page 15-28.

Code Prover does not consider atomicity of operations, so it continues to show `shared_var` as a potentially unprotected variable (the operations in `signal_handler_1` can still interrupt the operations in `signal_handler_2`). Code Prover shows `shared_var` as protected only when you specify both `signal_handler_1` and `signal_handler_2` as interrupts.

Protect By Disabling Interrupts

In a Bug Finder analysis, you can protect a group of operations by disabling all tasks and interrupts other than the current one.

Use the option `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)` to specify a routine that disables all interruption when called, and a routine that reenables them. The disabling routine disables preemption by all:

- Non-cyclic tasks.

  See `Tasks (-entry-points)`.
- Cyclic tasks.

  See `Cyclic tasks (-cyclic-tasks)`.
- Interrupts.

  See `Interrupts (-interrupts)`.

In other words, the analysis considers that the body of operations between the disabling routine and the enabling routine is atomic and not interruptible at all.

After you call a routine to disable interrupts, all subsequent operations are atomic until you call the other routine to reenable interrupts. The operations are atomic with respect to operations in all other tasks.

See Also

More About

- “Analyze Multitasking Programs in Polyspace” on page 15-2
- “Define Atomic Operations in Multitasking Code” on page 15-25

Define Atomic Operations in Multitasking Code

In code with multiple threads, you can use Polyspace Bug Finder to detect data races or Polyspace Code Prover to list potentially unprotected shared variables.

To determine if a variable shared between multiple threads is protected against concurrent access, Polyspace checks if the operations on the variable are atomic.

Nonatomic Operations

If an operation is nonatomic, Polyspace considers that the operation involves multiple steps. These steps do not need to occur together and can be interrupted by operations in other threads.

For instance, consider these two operations in two different threads:

- Thread 1: `var++;`

  This operation is nonatomic because it takes place in three steps: reading `var`, incrementing `var`, and writing back `var`.
- Thread 2: `var = 0;`

  This operation is atomic if the size of `var` is less than the word size on the target. See details below for how Polyspace determines the word size.

If the two operations are not protected (by using, for instance, critical sections), the operation in the second thread can interrupt the operation in the first thread. If the interruption happens after `var` is incremented in the first thread but before the incremented value is written back, you can see unexpected results.

What Polyspace Considers as Nonatomic

Code Prover considers all operations as nonatomic unless you protect them, for instance, by using critical sections. See “Define Specific Operations as Atomic”.

Bug Finder considers an operation as nonatomic if it can translate into more than one machine instruction. For instance:

- The operation can involve both a read and write operation. For example, `var++` involves reading the value of `var`, increasing the value by one and writing the increased value back to `var`.
- The operation can involve a 64-bit variable on a 32-bit target. For example, the operation

  ```
  long long var1, var2;
  var1=var2;
  ```

  involves two steps in copying the content of `var2` to `var1` on certain targets.

  Polyspace uses the **Pointer** size for your **Target processor type** as the threshold to compute atomicity. For instance, if you use `i386` as your **Target processor type**, the **Pointer** size is 32 bits and **Long long** and **Double** sizes are both 64 bits. Therefore, Polyspace considers copying one `long long` or `double` variable to another as nonatomic.

  See also `Target processor type (-target)`.

- The operation can involve writing the return value of a function call to a shared variable. For example, the operation `x=func()` involves calling `func` and writing the return value of `func` to `x`.

To detect data races where at least one of the two interrupting operations is nonatomic, enable the Bug Finder checker `Data race`. To remove this constraint on the checker, use the option `-detect-atomic-data-race`.

Define Specific Operations as Atomic

You might want to define a group of operations as atomic. This group of operations cannot be interrupted by operations in another thread or task.

Use one of these techniques:

- **Critical sections**

  Protect a group of operations with critical sections.

  A critical section begins and ends with calls to specific functions. You can use a predefined set of primitives to begin or end critical sections, or use your own functions.

  A group of operations in a critical section are atomic with respect to another group of operations that are in the same critical section (that is, having the same beginning and ending function).

  Specify critical sections using the option `Critical section details (-critical-section-begin -critical-section-end)`.
- **Temporally exclusive tasks**

  Protect a group of operations by specifying certain tasks as temporally exclusive.

  If a group of tasks are temporally exclusive, all operations in one task are atomic with respect to operations in the other tasks.

  Specify temporal exclusion using the option `Temporally exclusive tasks (-temporal-exclusions-file)`.
- **Task priorities**

  Protect a group of operations by specifying that certain tasks have higher priorities. For instance, interrupts have higher priorities over cyclic tasks.

  You can specify up to four different priorities with these options (with highest priority listed first):

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

  All operations in a task with higher priority are atomic with respect to operations in tasks with lower priorities. See also “Define Task Priorities for Data Race Detection in Polyspace” on page 15-28.
- **Routine disabling interrupts** (Bug Finder only)

Protect a group of operations by disabling all interrupts. Use the option `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

After you call a routine to disable interrupts, all subsequent operations are atomic until you call another routine to reenable interrupts. The operations are atomic with respect to operations in all other tasks.

For a tutorial, see "Protections for Shared Variables in Multitasking Code" on page 15-21.

See Also

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

More About

- "Analyze Multitasking Programs in Polyspace" on page 15-2
- "Protections for Shared Variables in Multitasking Code" on page 15-21

Define Task Priorities for Data Race Detection in Polyspace

Bug Finder detects data races between concurrent tasks. One of the ways you can fix data races is by specifying that certain tasks have higher priorities over others. A task with higher priority is atomic with respect to tasks with lower priority and cannot be interrupted by those tasks.

Emulating Task Priorities

You can specify up to four different priorities with these options (with highest priority listed first):

- Interrupts (nonpreemptable): Use option `Interrupts (-interrupts)`.
- Interrupts (preemptable): Use options `Interrupts (-interrupts)` and `-preemptable-interrupts`.
- Cyclic tasks (nonpreemptable): Use options `Cyclic tasks (-cyclic-tasks)` and `-non-preemptable-tasks`.

  You can also define preemptable noncyclic tasks with the option `Tasks (-entry-points)` and `-non-preemptable-tasks`.
- Cyclic tasks (preemptable): Use option `Cyclic tasks (-cyclic-tasks)`.

  You can also define noncyclic tasks with the option `Tasks (-entry-points)`.

For instance, interrupts have the highest priority and cannot be preempted by other tasks. To define a class of interrupts that can be preempted, lower their priority by making them preemptable.

Examples of Task Priorities

Consider this example with three tasks. A variable `var` is shared between the two tasks `task1` and `task2` without any protection such as a critical section. Depending on the priorities of `task1` and `task2`, Bug Finder shows a data race. The third task is not relevant for the example (and is added only to include a critical section, otherwise data race detection is disabled).

```
int var;

void begin_critical_section(void);
void end_critical_section(void);

void task1(void)  {
    var++;
}

void task2(void)  {
    var = 0;
}

void task3(void){
    begin_critical_section();
    /* Some atomic operation */
    end_critical_section();
}
```

Adjust the priorities of `task1` and `task2` and see whether a data race is detected. For instance:

1. Configure these multitasking options:

   - `Interrupts (-interrupts)`: Specify `task1` and `task2` as interrupts.
   - `Cyclic tasks (-cyclic-tasks)`: Specify `task3` as a cyclic task.
   - `Critical section details (-critical-section-begin -critical-section-end)`: Specify `begin_critical_section` as a function beginning a critical section and `end_critical_section` as a function ending a critical section.
2. Run Bug Finder.

   You do not see a data race. Since `task1` and `task2` are nonpreemptable interrupts, the shared variable cannot be accessed concurrently.
3. Change `task1` to a preemptable interrupt by using the option `-preemptable-interrupts`.
4. Run Bug Finder again. You now see a data race on the shared variable `var`.

Further Explorations

Modify this example in the following ways and see the effect of the modification:

- Change the priorities of `task1` and `task2`.

  For instance, you can leave `task1` as a nonpreemptable interrupt but change `task2` to a preemptable interrupt by using the option `-preemptable-interrupts`.

  The data race disappears. The reason is:

  - `task1` has higher priority and cannot be interrupted by `task2`.
  - The operation in `task2` is atomic and cannot be interrupted by `task1`.
- Specify the option `-detect-atomic-data-race`.

  You see the data race again. The checker considers all operations as potentially nonatomic and the operation in `task2` can now be interrupted by the higher priority operation in `task1`.

Try other modifications to the analysis options and see the result of the checkers.

Effect of Task Priorities in Code Prover

The options to specify task priorities are also accepted in Code Prover. However, Code Prover considers all operations as potentially non-atomic and interruptible. This overapproximation can lead to situations where the task priority specifications appear to be ignored.

For instance, in the preceding example, if you run Code Prover, the overapproximation can lead to false positives.

- If you specify both `task1` and `task2` as nonpreemptable interrupts, the shared variable `var` appears as a green **Shared protected global variable**. This is a sound result since both tasks cannot be interrupted.
- If you specify that `task1` has lower priority than `task2`, the shared variable `var` appears as an orange **Potentially unprotected variable**. This is a sound and precise result since the operation `var++` in `task1` is nonatomic and involves more than one machine instruction. The operation can be interrupted by the operation `var = 0` in `task2`.

- If you specify that `task1` has higher priority than `task2`, the shared variable `var` still appears as an orange **Potentially unprotected variable**. This is a sound but imprecise result:

  - The operation `var++` in `task1` cannot be interrupted because of the higher priority of `task1`.
  - The operation `var = 0` in `task2` cannot be interrupted because it is atomic.

  However, because Code Prover considers all operations as potentially non-atomic, it considers `var = 0` in `task2` as interruptible and therefore continues to show `var` as potentially unprotected.

See Also

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

Polyspace Results
`Data race` | `Potentially unprotected variable` | `Shared variable`

More About

- "Analyze Multitasking Programs in Polyspace" on page 15-2
- "Protections for Shared Variables in Multitasking Code" on page 15-21
- "Define Atomic Operations in Multitasking Code" on page 15-25

Define Critical Sections with Functions That Take Arguments

When verifying multitasking code, Polyspace considers that a critical section lies between calls to a lock and unlock function.

```
lock();
/* Critical section code */
unlock();
```

A group of operations in a critical section are atomic with respect to another group of operations that are in the same critical section (that is, having the same lock and unlock function). See also “Define Atomic Operations in Multitasking Code” on page 15-25.

Polyspace Assumption on Functions Defining Critical Sections

Polyspace ignores arguments to functions that begin and end critical sections.

For instance, Polyspace treats the two code sections below as the same critical section if you specify `my_task_1` and `my_task_2` as entry points, `my_lock` as the lock function and `my_unlock` as the unlock function.

```
int shared_var;

void my_lock(int);
void my_unlock(int);

void my_task_1() {
   my_lock(1);
   /* Critical section code */
   shared_var=0;
   my_unlock(1);
}

void my_task_2() {
   my_lock(2);
   /* Critical section code */
   shared_var++;
   my_unlock(2);
}
```

As a result, the analysis considers that these two sections are protected from interrupting each other even though they might not be protected. For instance, Bug Finder does not detect the data race on `shared_var`.

Often, the function arguments can be determined only at run time. Since Polyspace models the critical sections prior to the static analysis and run-time error checking phase, the analysis cannot determine if the function arguments are different and ignores the arguments.

Adapt Polyspace Analysis to Lock and Unlock Functions with Arguments

When the arguments to the functions defining critical sections are compile-time constants, you can adapt the analysis to work around the Polyspace assumption.

For instance, you can use Polyspace analysis options so that the code in the preceding example appears to Polyspace as shown here.

```
int shared_var;

void my_lock_1(void);
void my_lock_2(void);
void my_unlock_1(void);
void my_unlock_2(void);

void my_task_1() {
   my_lock_1();
   /* Critical section code */
   shared_var=0;
   my_unlock_1();
}

void my_task_2() {
   my_lock_2();
   /* Critical section code */
   shared_var++;
   my_unlock_2();
}
```

If you then specify `my_lock_1` and `my_lock_2` as the lock functions and `my_unlock_1` and `my_unlock_2` as the unlock functions, the analysis recognizes the two sections of code as part of different critical sections. For instance, Bug Finder detects a data race on `shared_var`.

To adapt the analysis for lock and unlock functions that take compile-time constants as arguments:

1. In a header file `common_polyspace_include.h`, convert the function arguments into extensions of the function name with `#define`-s. In addition, provide a declaration for the new functions.

   For instance, for the preceding example, use these `#define`-s and declarations:

   ```
   #define my_lock(X) my_lock_##X()
   #define my_unlock(X) my_unlock_##X()

   void my_lock_1(void);
   void my_lock_2(void);
   void my_unlock_1(void);
   void my_unlock_2(void);
   ```
2. Specify the file name `common_polyspace_include.h` as argument for the option `Include (-include)`.

   The analysis considers this header file as `#include`-d in all source files that are analyzed.
3. Specify the new function names as functions beginning and ending critical sections. Use the options `Critical section details (-critical-section-begin -critical-section-end)`.

See Also

`Critical section details (-critical-section-begin -critical-section-end)`

More About

- “Protections for Shared Variables in Multitasking Code” on page 15-21

Configure Coding Rules Checking and Code Metrics Computation

Check for and Review Coding Standard Violations

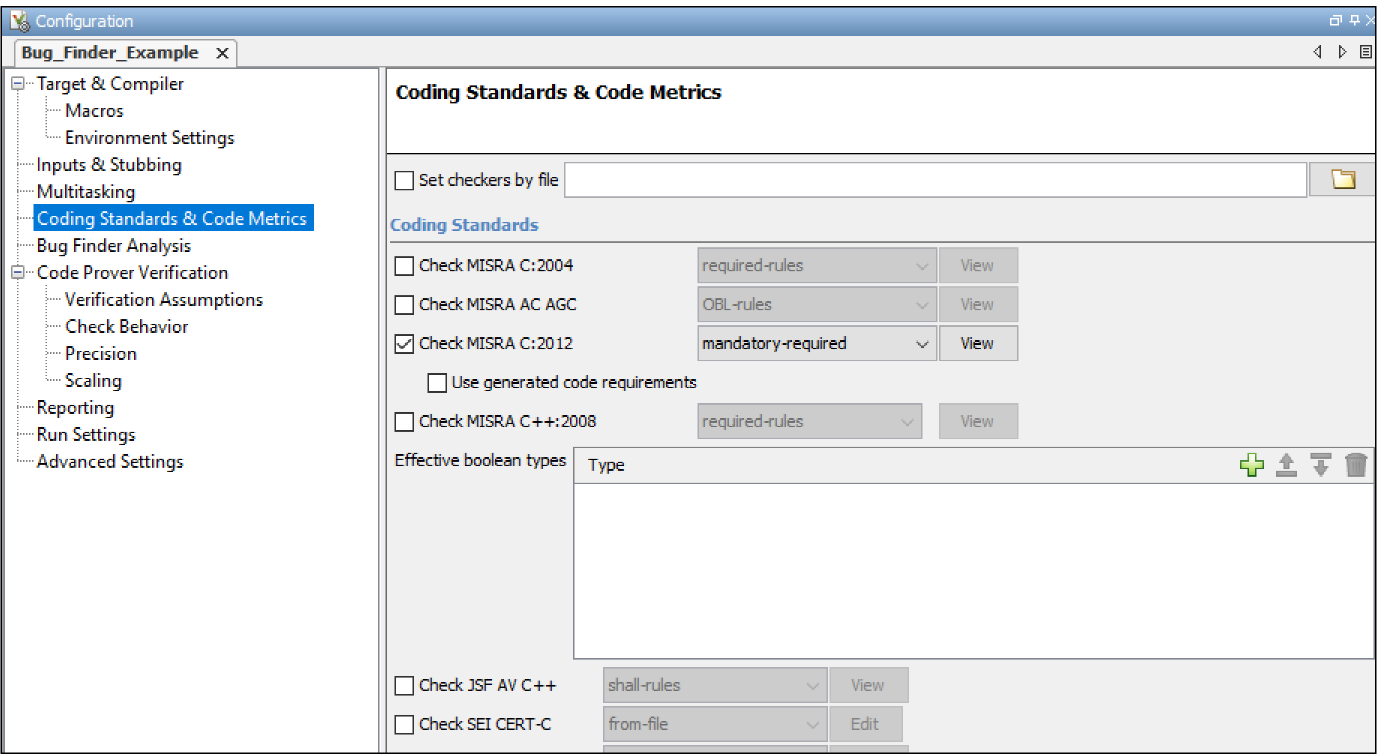
Note Starting in a future release, Code prover will not support checking coding rules. Migrate to Bug Finder for checking coding rules. See "Check for and Review Coding Standard Violations".

With Polyspace, you can check your C/C++ code for violations of coding rules such as MISRA C:2012 rules. Adhering to coding rules can reduce the number of defects and improve the quality of your code.

Polyspace can detect coding rule violations for these standards:

- MISRA C:2004
- MISRA C:2012
- MISRA C++
- JSF AV C++
- AUTOSAR C++14 *(Bug Finder only)*
- CERT C *(Bug Finder only)*
- CERT C++ *(Bug Finder only)*
- CWE™ *(Bug Finder only)*
- ISO®/IEC TS 17961 *(Bug Finder only)*
- Guidelines *(Bug Finder only)*

Configure Coding Rules Checking



Specify Standard and Predefined Checker Subsets

Specify the coding rules through Polyspace analysis options. When you run Bug Finder, the analysis looks for coding rule violations in addition to other checks. You can disable the other checks and look for coding rule violations only.

In the Polyspace user interface (desktop products), the options are on the **Configuration** pane under the **Coding Standards & Code Metrics** node.

For C code, use one of these options:

- `Check MISRA C:2004 (-misra2)`

  For generated code, enable the option specific to generated code.
- `Check MISRA C:2012 (-misra3)`

  For generated code, enable the option specific to generated code.
- `Check CWE (-cwe)`

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

- `Check MISRA C++:2008 (-misra-cpp)`
- `Check JSF AV C++ rules (-jsf-coding-rules)`
- `Check CWE (-cwe)`

You can specify a predefined subset of rules, for instance, `mandatory` for MISRA C:2012. These subsets are typically defined by the standard.

You can also define naming conventions for identifiers using regular expressions. See “Create Custom Coding Rules” on page 16-44.

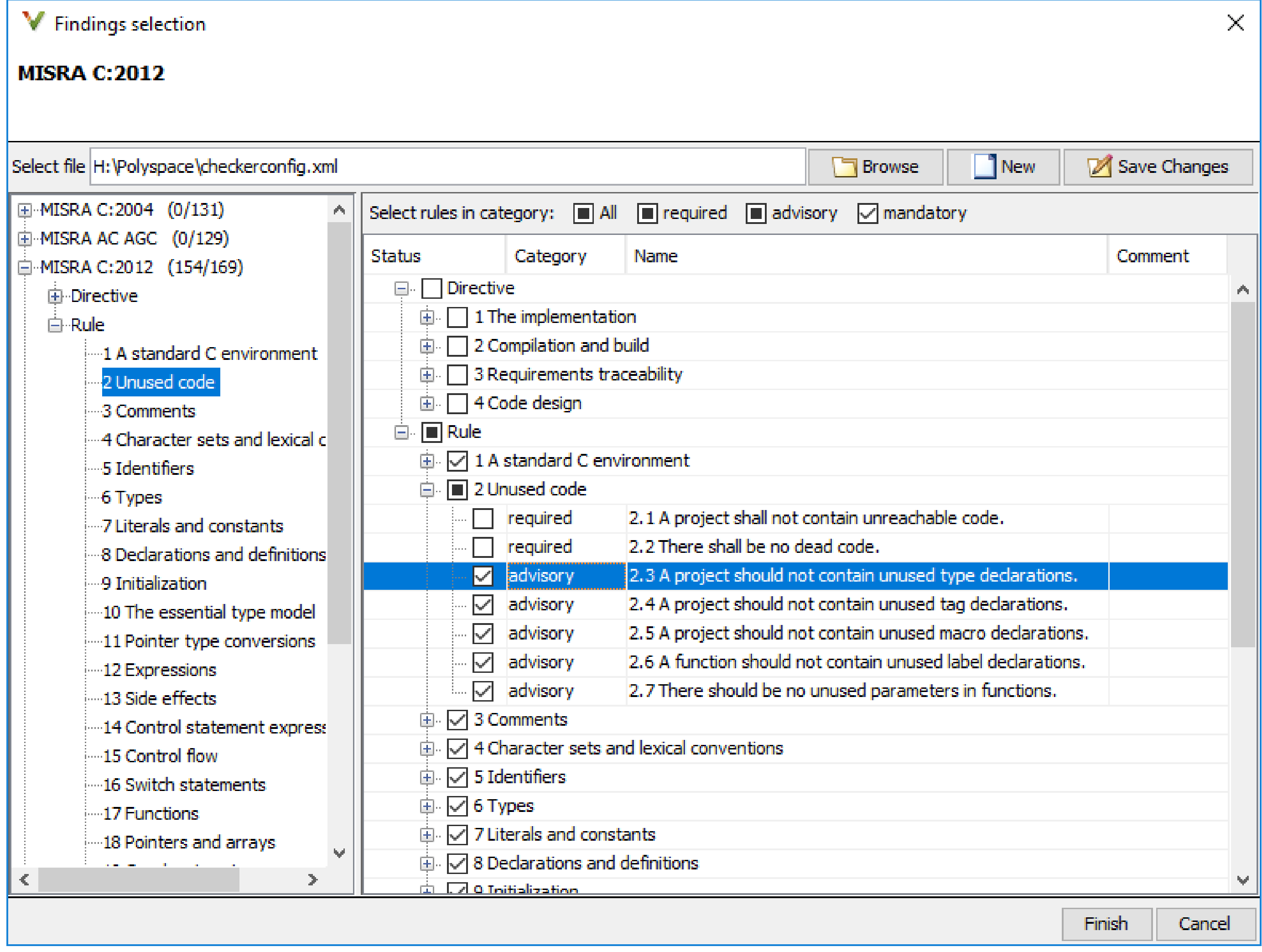
Customize Checker Subsets

Instead of the predefined subsets, you can specify your own subset of rules from a coding standard.

User Interface (Desktop Products Only)

1. Select the coding standard. From the drop-down list for the subset of rules, select `from-file`. Click **Edit**.
2. In the **Checkers selection** window, the coding standard is highlighted on the left pane. On the right pane, select the rules that you want to include in your analysis.

   - When selecting **Guidelines > Software Complexity** checkers, review their thresholds. If the default thresholds are not acceptable, specify a suitable threshold in the **Threshold** column. See `Check guidelines (-guidelines)`.
   - When selecting **Custom** rules, review the **Pattern** and **Convention** for the rules. See `Check custom rules (-custom-rules)`.



When you save the rule selections, the configuration is saved in an XML file that you can reuse for multiple analyses. The same file contains rules selected for all coding standards. You can reuse this file across multiple projects to enforce common coding standards in a team or organization. To reuse this file in another project in the Polyspace user interface:

- Choose a coding standard in the project configuration. From the drop-down list for the subset of rules, select `from-file`.
- Click **Edit** and browse to the file location. Alternatively, enter the file name as argument for the option `Set checkers by file (-checkers-selection-file)`.

Command Line

With the Polyspace desktop products, you can create a coding standard XML file in the user interface and then use this file for command-line analysis. Provide this XML file with the option `Set checkers by file (-checkers-selection-file)`.

With the Polyspace Server products, you have to create a coding standard XML from scratch. Depending on the standard that you want to enable, make a writeable copy of one of the files in *`polyspaceserverroot`*`\help\toolbox\bugfinder\examples\coding_standards_XML` and

turn off rules using entries in the XML file (all rules from a standard are enabled in the template). Here, *polyspaceserverroot* is the root installation folder for the Polyspace Server products, for instance, `C:\Program Files\Polyspace Server\R2023a`.

For instance, to turn off MISRA C:2012 rule 8.1, use this entry in a copy of the file `misra_c_2012_rules.xml`:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
      ...
      <check id="8.1" state="off">
      </check>
      ...
  </section>
  ...
</standard>
```

When using the Guideline checkers, specify their threshold between the `threshold` tags. For instance, to activates 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:

- “Custom Coding Rules”
- “Common Weakness Enumeration (CWE)”
- “JSF C++ Rules”
- “MISRA C:2004 Rules”
- “MISRA C:2012 Directives and Rules”
- “MISRA C++:2008 Rules”

Note The XML format of the checker configuration file can change in future releases.

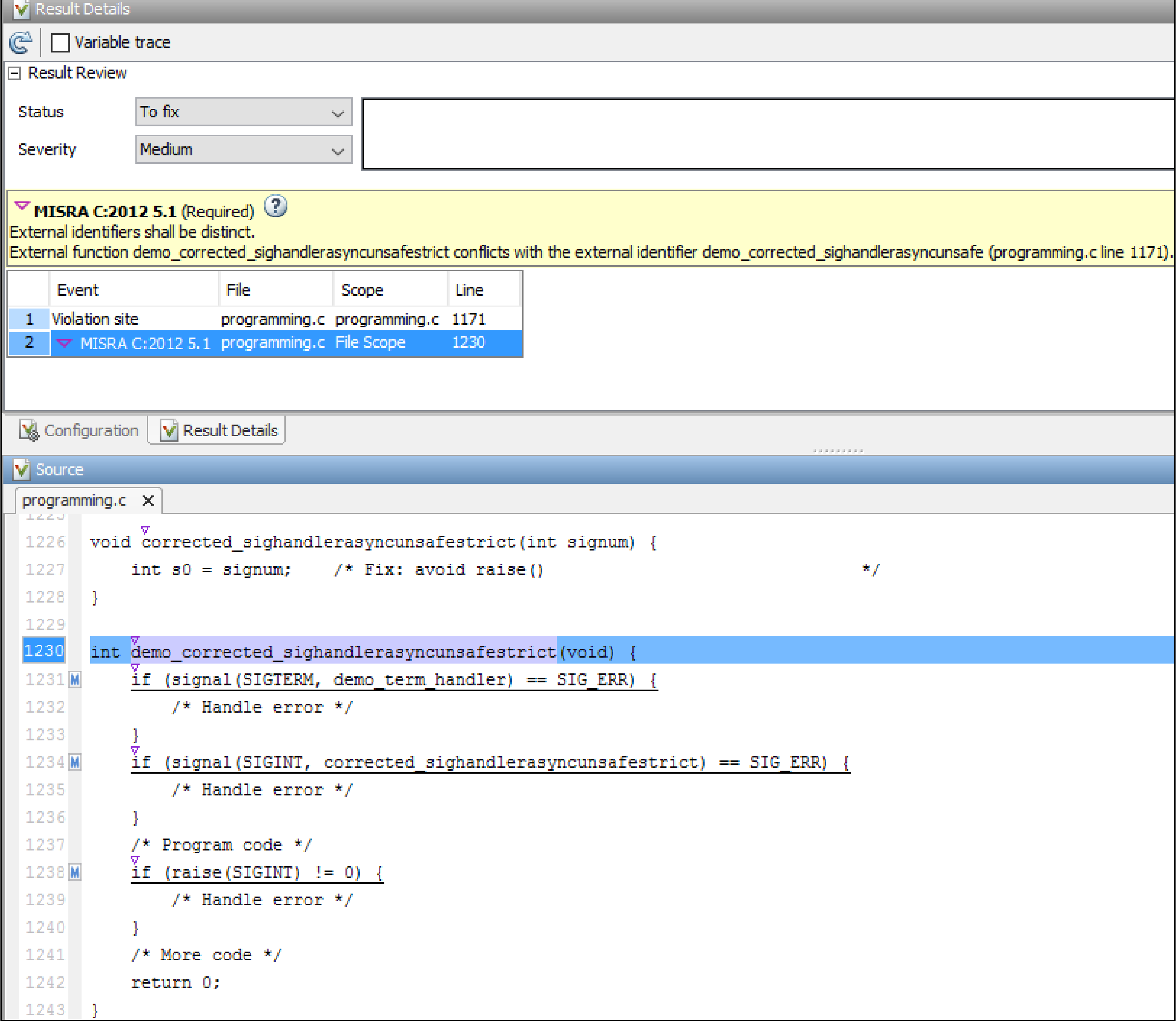
Check for Coding Standards Only

A Bug Finder analysis checks C/C++ code for:

- A default set of defects (bugs)
- Adherence to any coding standard that you specify.

To check for coding standards only, disable defect checking entirely. Specify `none` for the option `Find defects (-checkers)`.

Review Coding Rule Violations



After analysis, you see the coding standard violations on the **Results List** pane. Select a violation to see further details on the **Result Details** pane and the source code on the **Source** pane.

Violations of coding standards are indicated in the source code with the ▽ icon.

For further steps, see “Review Analysis Results” or “Review Polyspace Code Prover Results in Web Browser”.

Generate Reports

You can generate reports using templates that are explicitly defined for coding standards. Use the `CodingStandards` template. This template:

- Reports only coding standard violations in your analysis results, and omits other types of results such as defects, run-time errors or code metrics.
- Creates a separate chapter in the report for each coding standard. the chapter provides an overview of all violations of the standard and then lists each violation.

To specify a report template, use the option `Bug Finder and Code Prover report (-report-template)`.

See Also

More About

- "Interpret Code Prover Results in Polyspace Desktop User Interface" on page 21-2
- "Interpret Code Prover Results in Polyspace Access Web Interface" on page 26-2
- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" on page 23-2
- "Filter and Group Results in Polyspace Desktop User Interface" on page 24-2
- "Generate Reports from Polyspace Results" on page 25-2

Avoid Violations of MISRA C:2012 Rules 8.x

Note Starting in a future release, Code prover will not support checking MISRA C:2012 rules. Migrate to Bug Finder for checking these coding rules. See “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`.

Software Quality Objective Subsets (C:2004)

In this section...

"Rules in SQO-Subset1" on page 16-11
"Rules in SQO-Subset2" on page 16-12

Note Starting in a future release, Code Prover will not support checking compliance with external coding standards and calculating code metrics. Migrate to Bug Finder for these workflows. See "Software Quality Objective Subsets (C:2004)".

Rules in SQO-Subset1

The SQO subset1 consists of these rules:

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
8.11	The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
12.12	The underlying bit representations of floating-point values shall not be used.
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a for statement shall not contain any objects of floating type.
13.5	The three expressions of a for statement shall be concerned only with loop control.
14.4	The goto statement shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	The declaration of objects should contain no more than 2 levels of pointer indirection.

Rule number	Description
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.4	Unions shall not be used.
20.4	Dynamic heap memory allocation shall not be used.

Rules in SQO-Subset2

Good design practices generally lead to less code complexity. The following set of coding rules enforce good design practices. The `SQO-subset2` option checks the rules in `SQO-subset1` and some additional rules.

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
6.3	Typedefs that indicate size and signedness should be used in place of the basic types.
8.7	Objects shall be defined at block scope if they are only accessed from within a single function.
8.11	The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation.
9.2	Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures.
9.3	In an enumerator list, the '=' construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly initialised.
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.
10.5	If the bitwise operator ~ and << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
11.5	A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.
12.1	Limited dependence should be placed on C's operator precedence rules in expressions.
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits.

Rule number	Description
12.5	The operands of a logical && or || shall be primary-expressions.
12.6	The operands of a logical operators (&&, || and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, || and !).
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
12.10	The comma operator shall not be used.
12.12	The underlying bit representations of floating-point values shall not be used.
13.1	Assignment operators shall not be used in expressions that yield a Boolean value.
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a for statement shall not contain any objects of floating type.
13.5	The three expressions of a for statement shall be concerned only with loop control.
13.6	Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop.
14.4	The goto statement shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
14.8	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.
14.10	All if ... else if constructs should contain a final else clause.
15.3	The final clause of a switch statement shall be the default clause.
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.
16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesised parameter list, which may be empty.
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	The declaration of objects should contain no more than 2 levels of pointer indirection.

Rule number	Description
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.4	Unions shall not be used.
19.4	C macros shall only expand to a braced initialiser, a constant, a parenthesised expression,a type qualifier, a storage class specifier, or a do-while-zero construct.
19.9	Arguments to a function-like macro shall not contain tokens that look like pre-processing directives.
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.
19.12	There shall be at most one occurrence of the # or ## pre-processor operators in a single macro definition.
20.3	The validity of values passed to library functions shall be checked.
20.4	Dynamic heap memory allocation shall not be used.

Note Polyspace software does not check MISRA rule **20.3** directly.

However, you can check this rule by writing manual stubs that check the validity of values. For example, the following code checks the validity of an input being greater than 1:

```
int my_system_library_call(int in) {assert (in>1); if random \
return -1 else return 0; }
```

See Also

`Check MISRA C:2004 (-misra2)`

More About

- “Check for and Review Coding Standard Violations” on page 16-2

Software Quality Objective Subsets (AC AGC)

In this section...
“Rules in SQO-Subset1” on page 16-15
“Rules in SQO-Subset2” on page 16-16

Note Starting in a future release, Code Prover will not support checking compliance with external coding standards and calculating code metrics. Migrate to Bug Finder for these workflows. See “Software Quality Objective Subsets (AC AGC)”.

Rules in SQO-Subset1

The SQO subset1 consists of these rules:

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier
8.11	The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to object type or a pointer to void
11.3	A cast should not be performed between a pointer type and an integral type
12.12	The underlying bit representations of floating-point values shall not be used
13.4	The controlling expression of a for statement shall not contain any objects of floating type
13.5	The three expressions of a for statement shall be concerned only with loop control
14.4	The goto statement shall not be used
14.7	A function shall have a single point of exit at the end of the function
16.1	Functions shall not be defined with variable numbers of arguments
16.2	Functions shall not call themselves, either directly or indirectly
16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array
17.4	Array indexing shall be the only allowed form of pointer arithmetic
17.5	The declaration of objects should contain no more than 2 levels of pointer indirection
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist

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

For more information about these rules, see *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

Rules in SQO-Subset2

Good design practices generally lead to less code complexity. The following set of coding rules enforce good design practices. The `SQO-subset2` option checks the rules in `SQO-subset1` and some additional rules.

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier
6.3	Typedefs that indicate size and signedness should be used in place of the basic types
8.7	Objects shall be defined at block scope if they are only accessed from within a single function
8.11	The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation
9.2	Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures
9.3	In an enumerator list, the '=' construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly initialised
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression
10.5	If the bitwise operator ~ and << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to object type or a pointer to void
11.3	A cast should not be performed between a pointer type and an integral type
11.5	A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer
12.1	Limited dependence should be placed on C's operator precedence rules in expressions
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits

<table>
<tr><th>Rule number</th><th>Description</th></tr>
<tr><td>12.5</td><td>The operands of a logical && or || shall be primary-expressions</td></tr>
<tr><td>12.6</td><td>The operands of a logical operators (&&, || and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, || and !)</td></tr>
<tr><td>12.9</td><td>The unary minus operator shall not be applied to an expression whose underlying type is unsigned</td></tr>
<tr><td>12.10</td><td>The comma operator shall not be used</td></tr>
<tr><td>12.12</td><td>The underlying bit representations of floating-point values shall not be used</td></tr>
<tr><td>13.1</td><td>Assignment operators shall not be used in expressions that yield a Boolean value</td></tr>
<tr><td>13.2</td><td>Tests of a value against zero should be made explicit, unless the operand is effectively Boolean</td></tr>
<tr><td>13.4</td><td>The controlling expression of a for statement shall not contain any objects of floating type</td></tr>
<tr><td>13.5</td><td>The three expressions of a for statement shall be concerned only with loop control</td></tr>
<tr><td>13.6</td><td>Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop</td></tr>
<tr><td>14.4</td><td>The goto statement shall not be used</td></tr>
<tr><td>14.7</td><td>A function shall have a single point of exit at the end of the function</td></tr>
<tr><td>14.8</td><td>The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement</td></tr>
<tr><td>14.10</td><td>All if ... else if constructs should contain a final else clause</td></tr>
<tr><td>16.1</td><td>Functions shall not be defined with variable numbers of arguments</td></tr>
<tr><td>16.2</td><td>Functions shall not call themselves, either directly or indirectly</td></tr>
<tr><td>16.3</td><td>Identifiers shall be given for all of the parameters in a function prototype declaration</td></tr>
<tr><td>16.7</td><td>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</td></tr>
<tr><td>16.8</td><td>All exit paths from a function with non-void return type shall have an explicit return statement with an expression</td></tr>
<tr><td>16.9</td><td>A function identifier shall only be used with either a preceding &, or with a parenthesised parameter list, which may be empty</td></tr>
<tr><td>17.3</td><td>>, >=, <, <= shall not be applied to pointer types except where they point to the same array</td></tr>
<tr><td>17.4</td><td>Array indexing shall be the only allowed form of pointer arithmetic</td></tr>
<tr><td>17.5</td><td>The declaration of objects should contain no more than 2 levels of pointer indirection</td></tr>
<tr><td>17.6</td><td>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</td></tr>
<tr><td>18.4</td><td>Unions shall not be used</td></tr>
</table>

Rule number	Description
19.4	C macros shall only expand to a braced initialiser, a constant, a parenthesised expression,a type qualifier, a storage class specifier, or a do-while-zero construct
19.9	Arguments to a function-like macro shall not contain tokens that look like pre-processing directives
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator
19.12	There shall be at most one occurrence of the # or ## pre-processor operators in a single macro definition
20.3	The validity of values passed to library functions shall be checked
20.4	Dynamic heap memory allocation shall not be used

Note Polyspace software does not check MISRA rule **20.3** directly.

However, you can check this rule by writing manual stubs that check the validity of values. For example, the following code checks the validity of an input being greater than 1:

```
int my_system_library_call(int in) {assert (in>1); if random \
return -1 else return 0; }
```

For more information about these rules, see *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

See Also

`Check MISRA AC AGC (-misra-ac-agc)`

More About

- “Check for and Review Coding Standard Violations” on page 16-2

Software Quality Objective Subsets (C:2012)

In this section...
“Guidelines in SQO-Subset1” on page 16-19
“Guidelines in SQO-Subset2” on page 16-20

Note Starting in a future release, Code Prover will not support checking compliance with external coding standards and calculating code metrics. Migrate to Bug Finder for these workflows. See “Software Quality Objective Subsets (C:2012)”.

These subsets of MISRA C:2012 guidelines can have a direct or indirect impact on the precision of your Polyspace results. When you set up coding rules checking, you can select these subsets.

Guidelines in SQO-Subset1

The SQO subset1 consists of these rules:

Rule	Description
D1.1	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood.
5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.
8.11	When an array with external linkage is declared, its size should be explicitly specified.
8.13	A pointer should point to a const-qualified type whenever possible.
11.1	Conversions shall not be performed between a pointer to a function and any other type.
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type.
11.4	A conversion should not be performed between a pointer to object and an integer type.
11.5	A conversion should not be performed from pointer to void into pointer to object.
11.6	A cast shall not be performed between pointer to void and an arithmetic type.
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type.
14.1	A loop counter shall not have essentially floating type.
14.2	A for loop shall be well-formed.
15.1	The goto statement should not be used.
15.2	The goto statement shall jump to a label declared later in the same function.

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.
17.1	The features of <stdarg.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.
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. The following set of coding rules enforce good design practices. The `SQO-subset2` option checks the rules in `SQO-subset1` and some additional rules.

Rule	Description
D1.1	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood.
D4.6	typedefs that indicate size and signedness should be used in place of the basic numerical types.
D4.11	The validity of values passed to library functions shall be checked.
5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
8.2	Function types shall be in prototype form with named parameters.
8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.
8.9	An object should be defined at block scope if its identifier only appears in a single function.
8.11	When an array with external linkage is declared, its size should be explicitly specified.
8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.
8.13	A pointer should point to a const-qualified type whenever possible.
9.2	The initializer for an aggregate or union shall be enclosed in braces.
9.3	Arrays shall not be partially initialized.

Rule	Description
10.1	Operands shall not be of an inappropriate essential type.
10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type.
11.1	Conversions shall not be performed between a pointer to a function and any other type.
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type.
11.4	A conversion should not be performed between a pointer to object and an integer type.
11.5	A conversion should not be performed from pointer to void into pointer to object.
11.6	A cast shall not be performed between pointer to void and an arithmetic type.
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type.
11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.
12.1	The precedence of operators within expressions should be made explicit.
12.3	The comma operator should not be used
13.2	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.
13.4	The result of an assignment operator should not be used.
14.1	A loop counter shall not have essentially floating type.
14.2	A for loop shall be well-formed.
14.4	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type.
15.1	The goto statement should not be used.
15.2	The goto statement shall jump to a label declared later in the same function.
15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.
15.5	A function should have a single point of exit at the end.
15.6	The body of an iteration-statement or a selection-statement shall be a compound-statement.
15.7	All if … else if constructs shall be terminated with an else statement.
16.4	Every switch statement shall have a default label.
16.5	A default label shall appear as either the first or the last switch label of a switch statement.
17.1	The features of <stdarg.h> shall not be used.
17.2	Functions shall not call themselves, either directly or indirectly.
17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Rule	Description
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 and Review Coding Standard Violations” on page 16-2

Software Quality Objective Subsets (C++)

In this section...
“SQO Subset 1 – Direct Impact on Selectivity” on page 16-23
“SQO Subset 2 – Indirect Impact on Selectivity” on page 16-24

Note Starting in a future release, Code Prover will not support checking compliance with external coding standards and calculating code metrics. Migrate to Bug Finder for these workflows. See “Software Quality Objective Subsets (C++)”.

SQO Subset 1 – Direct Impact on Selectivity

The SQO subset 1 consists of these MISRA C++:2008 rules:

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.
3-9-3	The underlying bit representations of floating-point values shall not be used.
5-0-15	Array indexing shall be the only form of pointer arithmetic.
5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.

MISRA C++ Rule	Description
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.
9-5-1	Unions shall not be used.
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.
10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy.
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base clas s.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
18-4-1	Dynamic heap memory allocation shall not be used.

SQO Subset 2 – Indirect Impact on Selectivity

Good design practices generally lead to less code complexity. The following set of coding rules may help to address design issues in your code. The `SQO-subset2` option checks the rules in `SQO-subset1` and `SQO-subset2`.

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
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.

MISRA C++ Rule	Description
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.
5-2-11	The comma operator, && operator and the || operator shall not be overloaded.
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.

MISRA C++ Rule	Description
8-5-3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.
9-5-1	Unions shall not be used.
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.
10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy.
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.

MISRA C++ Rule	Description
16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition.
18-4-1	Dynamic heap memory allocation shall not be used.

See Also

`Check MISRA C++:2008 (-misra-cpp)`

More About

- “Check for and Review Coding Standard Violations” on page 16-2

Coding Rule Subsets Checked Early in Analysis

Note Starting in a future release, Code prover will not support checking coding rules. Migrate to Bug Finder for checking coding rules. See "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 and Review Coding Standard Violations" on page 16-2.

MISRA C:2004 and MISRA AC AGC Rules

The software checks the following rules early in the analysis. The rules that are checked at a system level and appear only in the `system-decidable-rules` subset are indicated by an asterisk.

Environment

Rule	Description
1.1*	All code shall conform to ISO 9899:1990 "Programming languages - C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.

Language Extensions

Rule	Description
2.1	Assembly language shall be encapsulated and isolated.
2.2	Source code shall only use `/* */` style comments.
2.3	The character sequence `/*` shall not be used within a comment.

Documentation

Rule	Description
3.4	All uses of the `#pragma` directive shall be documented and explained.

Character Sets

Rule	Description
4.1	Only those escape sequences which are defined in the ISO C standard shall be used.
4.2	Trigraphs shall not be used.

Identifiers

Rule	Description
5.1*	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
5.3*	A typedef name shall be a unique identifier.
5.4*	A tag name shall be a unique identifier.
5.5*	No object or function identifier with a static storage duration should be reused.
5.6*	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.
5.7*	No identifier name should be reused.

Types

Rule	Description
6.1	The plain char type shall be used only for the storage and use of character values.
6.2	Signed and unsigned char type shall be used only for the storage and use of numeric values.
6.3	`typedefs` that indicate size and signedness should be used in place of the basic types.
6.4	Bit fields shall only be defined to be of type `unsigned int` or `signed int`.
6.5	Bit fields of type `signed int` 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 `static` storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by 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 = 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: • 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 • 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 ~ and << are applied to an operand of underlying type `unsigned char` or `unsigned short`, the result shall be immediately cast to the underlying type of the operand
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 `void`.
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 `const` or `volatile` 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 `sizeof` operator should not be used on expressions that contain side effects.
12.5	The operands of a logical && or `||` shall be primary-expressions.
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 `!`).
12.7	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 (`++`) and decrement (`--`) 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 `for` statement shall not contain any objects of floating type.
13.5	The three expressions of a `for` statement shall be concerned only with loop control.
13.6	Numeric variables being used within a `for` loop for iteration counting should not be modified in the body of the loop.

Control Flow

Rule	Description
14.3	All non-null statements shall either • have at least one side effect however executed, or • cause control flow to change.
14.4	The `goto` statement shall not be used.
14.5	The `continue` statement shall not be used.
14.6	For any iteration statement, there shall be at most one `break` statement used for loop termination.
14.7	A function shall have a single point of exit at the end of the function.
14.8	The statement forming the body of a `switch`, `while`, `do while` or `for` statement shall be a compound statement.
14.9	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 `if else if` constructs should contain a final `else` clause.

Switch Statements

Rule	Description
15.0	Unreachable code is detected between `switch` statement and first `case`.
15.1	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 `break` statement shall terminate every non-empty `switch` clause.
15.3	The final clause of a `switch` statement shall be the `default` clause.
15.4	A `switch` expression should not represent a value that is effectively Boolean.
15.5	Every `switch` statement shall have at least one `case` 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 `void`.
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 &, 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	`#include` 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 `#include` directives.
19.3	The `#include` directive shall be followed by either a <filename> 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 `#defined` and `#undefd` within a block.
19.6	`#undef` 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 # 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.
19.13	The # and ## preprocessor operators should not be used.
19.14	The defined 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 `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` 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 `errno` shall not be used.
20.6	The macro `offsetof`, in library `<stddef.h>`, shall not be used.
20.7	The `setjmp` macro and the `longjmp` function shall not be used.
20.8	The signal handling facilities of `<signal.h>` shall not be used.
20.9	The input/output library `<stdio.h>` shall not be used in production code.
20.10	The library functions `atof`, `atoi` and `atoll` from library `<stdlib.h>` shall not be used.
20.11	The library functions `abort`, `exit`, `getenv` and `system` from library `<stdlib.h>` shall not be used.
20.12	The time handling functions of library `<time.h>` 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 `/*` and `//` shall not be used within a comment.
3.2	Line-splicing shall not be used in `//` 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 `static` storage class specifier shall be used in all declarations of objects and functions that have internal linkage.
8.9*	An object should be defined at block scope if its identifier only appears in a single function.
8.10	An inline function shall be declared with the `static` 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 `restrict` 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 `switch` statements shall be well-formed.
16.2	A `switch` label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement.
16.3	An unconditional `break` statement shall terminate every `switch`-clause.
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.
16.6	Every `switch` statement shall have at least two `switch`-clauses.
16.7	A `switch`-expression shall not have essentially Boolean type.

Functions

Rule	Description
17.1	The features of <starg.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 static 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 union keyword should not be used.

Preprocessing Directives

Rule	Description
20.1	#include directives should only be preceded by preprocessor directives or comments.
20.2	The ', ", or \ characters and the /* or // character sequences shall not occur in a header file name.
20.3	The #include directive shall be followed by either a <filename> or \"filename\" sequence.
20.4	A macro shall not be defined with the same name as a keyword.
20.5	#undef 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 #if or #elif preprocessing directive shall evaluate to 0 or 1.
20.9	All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation.
20.10	The # and ## preprocessor operators should not be used.
20.11	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.
20.12	A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.
20.13	A line whose first token is # shall be a valid preprocessing directive.
20.14	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related.

Standard Libraries

Rule	Description
21.1	#define and #undef 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 <stdlib.h> shall not be used.
21.4	The standard header file <setjmp.h> shall not be used.
21.5	The standard header file <signal.h> shall not be used.
21.6	The Standard Library input/output functions shall not be used.
21.7	The atof, atoi, atol, and atoll functions of <stdlib.h> shall not be used.
21.8	The library functions abort, exit, getenv and system of <stdlib.h> shall not be used.
21.9	The library functions bsearch and qsort of <stdlib.h> shall not be used.
21.10	The Standard Library time and date functions shall not be used.
21.11	The standard header file <tgmath.h> shall not be used.
21.12	The exception handling features of <fenv.h> should not be used.

The rules that are checked at a system level and appear only in the `system-decidable-rules` subset are indicated by an asterisk.

See Also

`Check MISRA C:2004 (-misra2)` | `Check MISRA AC AGC (-misra-ac-agc)` | `Check MISRA C:2012 (-misra3)`

More About

- "Check for and Review Coding Standard Violations" on page 16-2

Create Custom Coding Rules

Note Starting in a future release, Code prover will not support checking naming conventions. Migrate to Bug Finder for enforcing naming convention. See “Create Custom Coding Rules”.

This example shows how to check for violations of naming conventions on functions and objects in your C/C++ code. For each naming convention, you specify a pattern in the form of a regular expression. The software compares the pattern to identifiers in the source code and determines whether the identifiers follow those naming conventions.

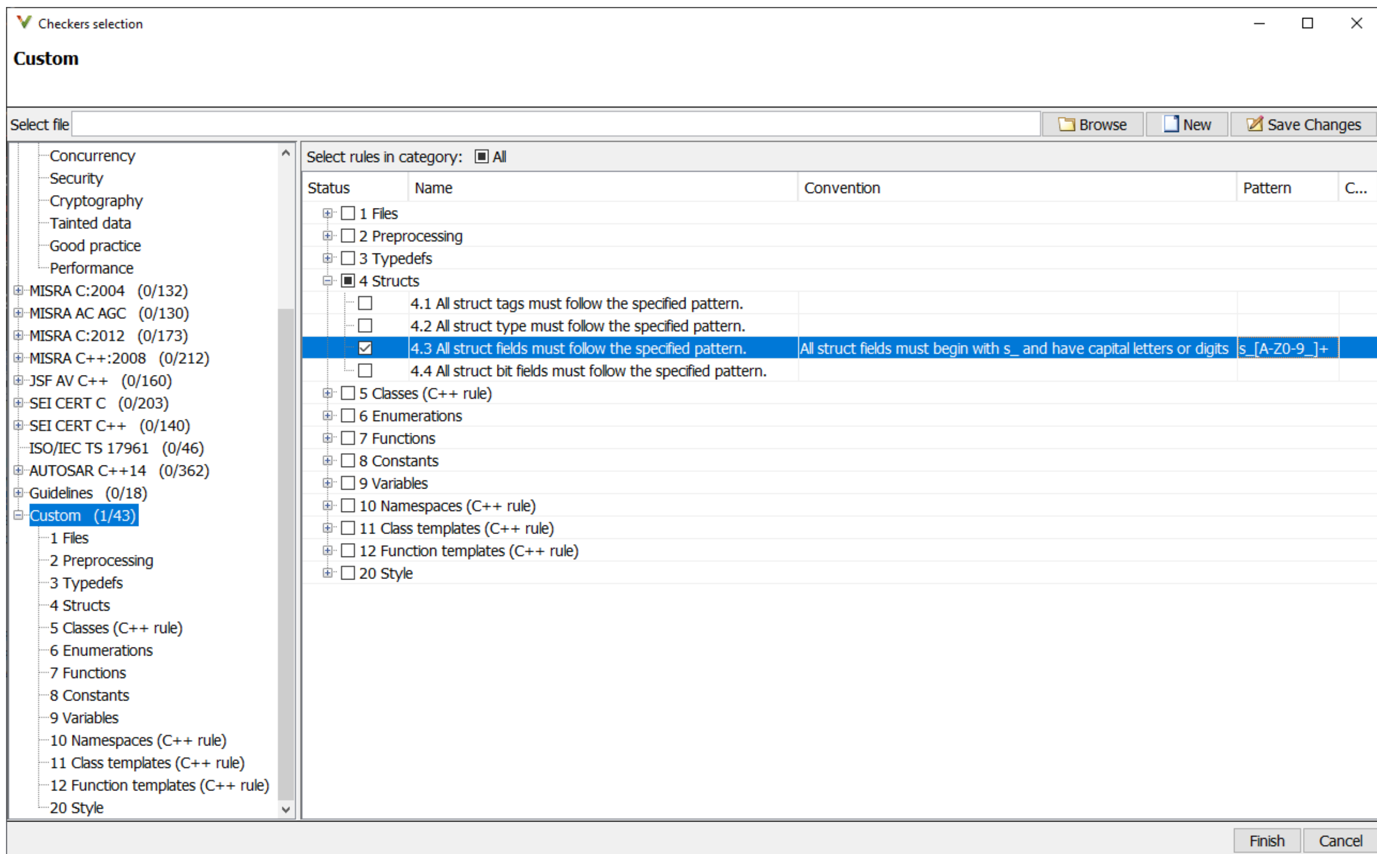
The tutorial uses this code stored in a file `printInitialValue.c`:

```
#include <stdio.h>

typedef struct {
    int a;
    int b;
} collection;

void main()
{
    collection myCollection= {0,0};
    printf("Initial values in the collection are %d and %d.",
            myCollection.a,myCollection.b);
}
```

Specify Naming Convention

Custom coding rule checkers compare the identifiers in your code to a naming convention that you specify. Polyspace raises a violation if the identifiers do not match the convention. Before you use the custom coding rules to enforce the naming convention, specify the naming convention by using a regular expression.

1. Open the Checkers Selection window. Depending on your workflow, you might open the window by using one of the Polyspace as You Code IDE plugins, the desktop user interface, or the command `polyspace-checkers-selection`.
2. In the Checkers Selection window, select the rule `4.3`.
3. In the **Convention** field, enter the message that you want to display when the rule is violated. This message describes the naming convention that you want to specify. For instance, in the **Convention** field, enter `All struct fields must begin with s_ and have capital letters or digits`.
4. In the **Pattern** field, enter a regular expression corresponding to the naming convention that you want to specify. For instance, to represent `struct` field names that begin with `s_` and have capital letters or digits, specify `s_[A-Z0-9_]+`. Polyspace supports Perl regular expressions when defining patterns. See `Check custom rules (-custom-rules)`.



A custom rule is not activated if the **Pattern** field is empty.

5. The **Comment** field is optional. A comment does not appear in the Polyspace results list. Leave the **Comment** field blank.
6. Save your changes in an XML file and close the window. This XML file can be used to check the specified custom rule.

Alternatively, edit a preexisting checkers XML file to specify naming conventions. The Polyspace installation folder contains a template that you can copy and edit.

1. Locate the template `custom_rules.xml` in *`polyspaceroot`*`\help\toolbox\bugfinder\examples\coding_standards_XML`. Here, *`polyspaceroot`* is the root installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace Server\R2023a`. Make an editable copy of the file `custom_rules.xml`.
2. In the editable XML file, locate the node corresponding to rule 4.3. Set the `state` attribute to `on`. Add a subnode `Convention` and specify it as `All struct fields must begin with s_ and have capital letters or digits`. Then, add a subnode `Pattern` and specify it as `s_[A-Z0-9_]+`. For instance,

```
<check id="4.3" state="on">
<convention>All struct fields must begin with s_
and have capital letters or digits</convention>
<pattern>s_[A-Z0-9_]+</pattern>
</check>
```

3. Save the XML file. You can use this XML file check the specified custom rule.

Check for Violations of Defined Custom Coding Rule

After specifying the naming convention, run a Polyspace analysis.

- If you are using the Polyspace desktop UI or one of the Polyspace as You Code plugins in an IDE, run a Polyspace analysis after saving your changes in the Checkers Selection window.
- If you are using the command line interface, provide the modified `custom_rules.xml` file as the argument for the option during analysis, along with the option . For instance, for custom rules checking by using Polyspace Bug Finder Server, enter:

```
polyspace-bug-finder-server -sources printInitialValue.c -custom-rules from-file
                    -checkers-selection-file custom_rules.xml
```

The Polyspace analysis reports two violations of custom rule 4.3 on the two fields `collection.a` and `collection.b`.

To resolve the defects, change the name of the fields so that they comply with the naming convention. For instance, rename the fields as `s_A` and `s_B` respectively. After renaming the fields, run another Polyspace analysis to verify that the naming convention is no longer violated.

See Also

More About

- "Setting Checkers in Polyspace as You Code"
- "Run Polyspace Bug Finder on Desktop"
- Perl Regular Expression

Compute Code Complexity Metrics Using Polyspace

Note Starting in a future release, Code prover will not support calculating code metrics. Migrate to Bug Finder for calculating code metrics. See "Compute Code Complexity Metrics Using Polyspace".

Code complexity metrics are a set of numbers that quantify the complexity of your C/C++ program. For instance:

- A function with a high cyclomatic complexity contains too many branches.
- A function with a high number of `return` statements has too many exit points.

Complex programs are difficult to debug, analyze, test and maintain. To avoid too much complexity, impose limits on the complexity metrics during coding.

Polyspace does not compute code complexity metrics by default. To compute them during analysis, use the option `Calculate code metrics (-code-metrics)`.

After analysis, the software displays project, file and function metrics on the **Results List** pane. You can compare the computed metric values against predefined limits. If a metric value exceeds limits, you can redesign your code to lower the metric value. For instance, if the number of called functions is high and several of those functions are always called together, you can write one function that fuses the bodies of those functions. Call that one function instead of the group of functions that are called together.

Impose Limits on Metrics (Desktop Products Only)

In the user interface of the Polyspace desktop products, open some results with metrics computations. Then impose limits on the metric values and update results on the **Results List** pane to show only metric values that exceed the limits.

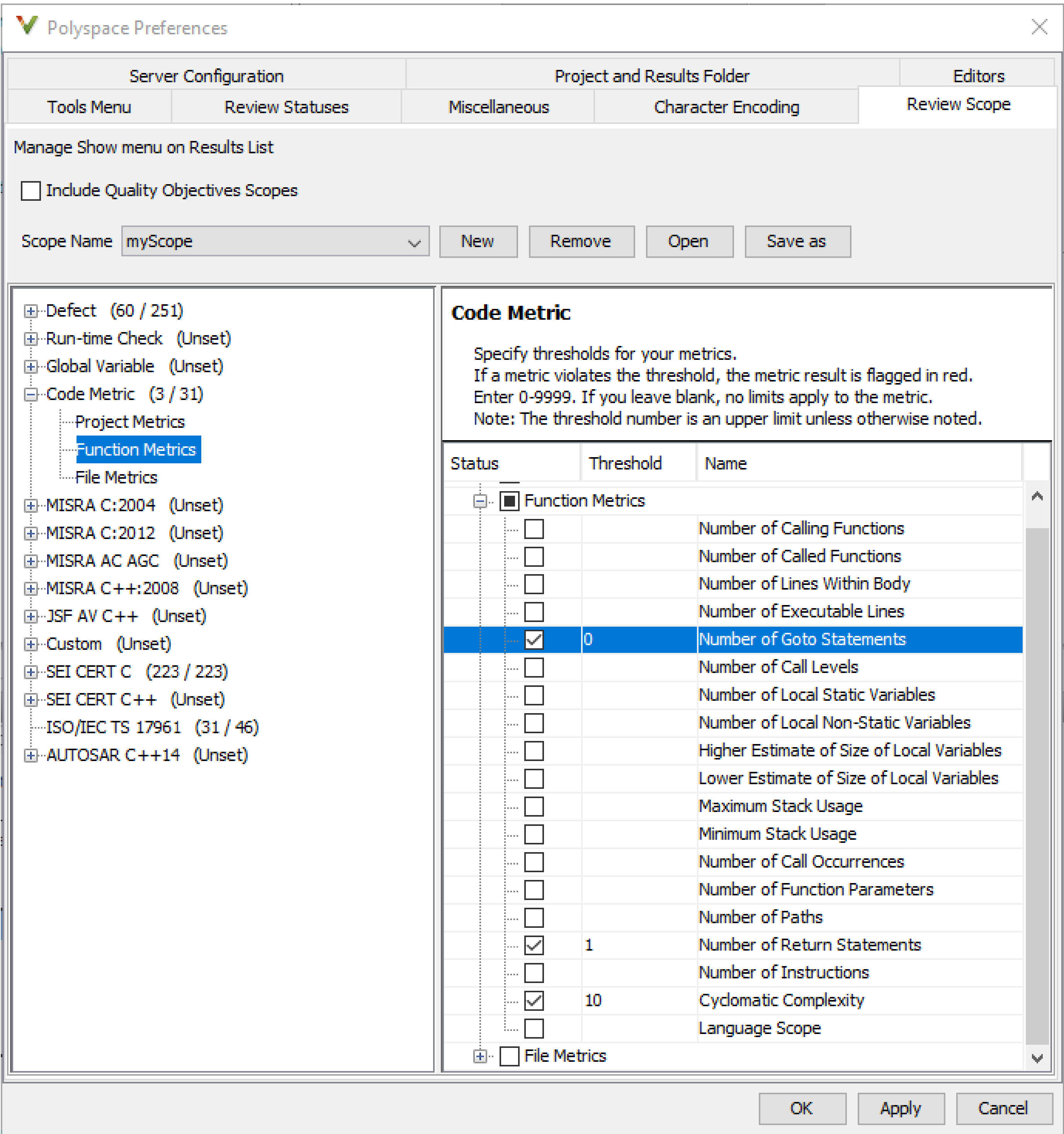
1. Select **Tools** > **Preferences**.
2. On the **Review Scope** tab, do one of the following:

   - To use a predefined limit, select **Include Quality Objectives Scopes**.

     The **Scope Name** list shows the additional option `HIS`. The option `HIS` displays the HIS code metrics on page 16-50 only. Select the option to see the limit values.
   - To define your own limits, select **New**. Save your limits file.

     On the left pane, select **Code Metric**. On the right, select a metric and specify a limit value for the metric. Other than **Comment Density**, limit values are upper limits.

     To select all metrics in a category such as **Function Metrics**, select the box next to the category name. For more information on the metrics categories, see "Code Metrics". If only a some metrics in a category are selected, the check box next to the category name displays a ▣ symbol.



3 Select **Apply** or **OK**.

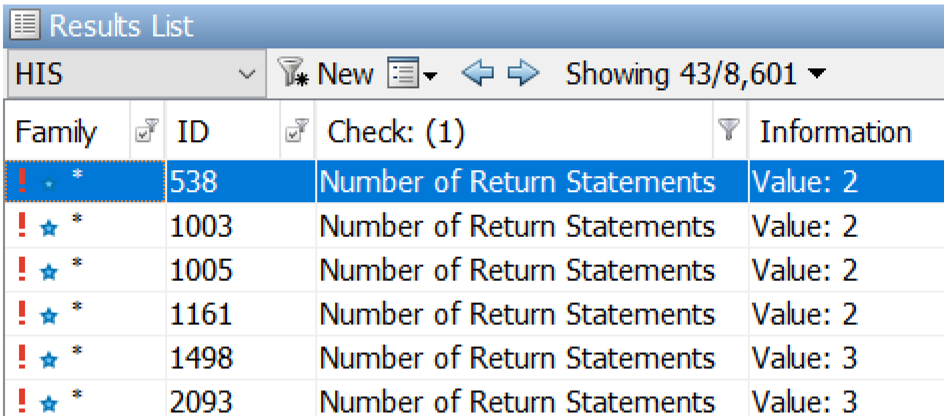
The drop-down list in the left of the **Results List** pane toolbar displays additional options.

- If you use predefined limits, the option `HIS` appears. This option displays code metrics only.

- If you define your own limits, the option corresponding to your limits file name appears.

4 Select the option corresponding to the limits that you want. Only metric values that violate your limits remain on the **Results List** pane.

   These metrics are shown along with a red exclamation mark. For instance, the predefined scope, `HIS`, requires that every function should have only one return statement. If you select the scope `HIS`, you see the metric **Number of return statements** only if the number exceeds one.



5 Review each violation and decide how to rework your code to avoid the violation.

Note To enforce coding standards across your organization, share your limits file that you saved in XML format.

People in your organization can use the **Open** button on the **Review Scope** tab and navigate to the location of the XML file.

Impose Limits on Metrics (Server and Access products)

In the Polyspace Access web interface, limits on code complexity metrics are predefined. In the **Dashboard** perspective, if you select **Code Metric**, a **Code Metrics** window shows the metric values and limits.

To find the limits used, see “HIS Code Complexity Metrics” on page 16-50.

See also “Code Metrics Dashboard in Polyspace Access Web Interface” on page 26-9.

See Also

`Calculate code metrics (-code-metrics)`

More About

- “Code Metrics”
- “HIS Code Complexity Metrics” on page 16-50

HIS Code Complexity Metrics

Note Starting in a future release, Code prover will not support calculating code metrics. Migrate to Bug Finder for calculating code metrics. See “HIS Code Complexity Metrics”.

The following list shows the Hersteller Initiative Software (HIS) standard metrics that Polyspace evaluates. These metrics and the recommended limits for their values are part of a standard defined by a major group of Original Equipment Manufacturers or OEMs. For more information on how to focus your review to this subset of code metrics, see “Compute Code Complexity Metrics Using Polyspace” on page 16-47.

Project

Polyspace evaluates the following HIS metrics at the project level.

Metric	Recommended Upper Limit
Number of direct recursions (`AP_CG_DIRECT_CYCLE`)	0
Number of recursions (`AP_CG_CYCLE`)	0

File

Polyspace evaluates the HIS metric, comment density, at the file level. The recommended lower limit is 20.

Function

Polyspace evaluates the following HIS metrics at the function level.

Metric	Recommended Upper Limit
Cyclomatic complexity (`VG`)	10
Language scope (`VOCF`)	4
Number of call levels (`LEVEL`)	4
Number of calling functions (`CALLING`)	5
Number of called functions (`CALLS`)	7
Number of function parameters (`PARAM`)	5
Number of goto statements (`GOTO`)	0
Number of instructions (`STMT`)	50
Number of paths (`PATH`)	80
Number of return statements (`RETURN`)	1

See Also

More About

- “Compute Code Complexity Metrics Using Polyspace” on page 16-47
- “Code Metrics”

Migrate Code Prover Workflows for Checking Coding Standards and Code Metrics to Bug Finder

In previous releases, Polyspace Code Prover supported checking of external coding standards and computation of code complexity metrics. For instance, you might be using Code Prover to:

- Check compliance with external coding standards such as MISRA C:2012 or MISRA C++:2008.
- Check compliance with naming conventions.
- Check compliance with code complexity standards.
- Calculate code metrics.

Support for the preceding capabilities will be removed from Code Prover in a future release. Starting in R2021b, Polyspace Bug Finder is the recommended tool for performing these tasks. Modify your workflows to migrate from using Code Prover to using Bug Finder.

Changes in Workflow

To migrate from Code Prover to Bug Finder, your workflow might need some changes.

Check for Coding Rule Violations and Compute Code Metrics

Previously, to perform this task, you configured the appropriate options in the **Configuration** pane, and then clicked **Run Code Prover**. To migrate to using Bug Finder:

- In the Polyspace user interface, configure the same options in the **Configuration** pane, and then click **Run Bug Finder**.
- At the command line, replace `polyspace-code-prover` by `polyspace-bug-finder`. If you do not want to enable the Bug Finder defects, specify `-checkers` with the value `none`. For instance, replace this command:

  ```
  polyspace-code-prover -sources file_name -misra3 all -code-metrics
  ```

  with this command:

  ```
  polyspace-bug-finder -sources file_name -misra3 all -code-metrics -checkers none
  ```

Polyspace Bug Finder checks some coding rules differently compared to Code Prover. After migrating to Bug Finder, you might see some small difference in the number and location of coding rule violations.

Compute Code Metrics, Check for Run-Time Errors and Coding Rule Violations

Previously, to perform these tasks, you configured the appropriate options in the **Configuration** pane, and then clicked **Run Code Prover**. To migrate to using Bug Finder:

- In the Polyspace user interface, configure the same options in the **Configuration** pane. Then, obtain these results by performing two separate Polyspace analyses. Run a Bug Finder analysis to check for coding rule violations and to compute code metrics. Run a separate Code Prover verification to check for run-time errors.
- At the command line, run separate Bug Finder and Code Prover analyses by using the commands `polyspace-bug-finder` and `polyspace-code-prover` with appropriate analysis options. For instance, replace this command:

```
polyspace-code-prover -sources file_name -misra3 all -code-metrics
```

with this command:

```
polyspace-bug-finder -sources file_name -code-metrics -misra3 all -checkers none
polyspace-code-prover -sources file_name
```

Compute Code Metrics, Check for Run-Time Errors and Coding Rule Violations in Generated Code

Previously, to perform these tasks, you configured your Polyspace analysis, and then started a Code Prover verification. To migrate to using Bug Finder:

- On the Simulink toolstrip, use the same configurations that you used before. Then, run separate Bug Finder and Code Prover analyses.
- In the MATLAB Command Window, use separate sets of `polyspace.ModelLinkOptions` and `polyspace.Project` objects to perform separate Bug Finder and Code Prover analyses.

Sample MATLAB Code

```
% Make directory for code generation
[TEMPDIR, CGDIR] = rtwdemodir();
%  Specify model name
modelName = 'rtwdemo_roll';
% Load the model
load_system(modelName);

%  Set parameters for Embedded Coder target
set_param(modelName, 'SystemTargetFile', 'ert.tlc');
set_param(modelName,'Solver','FixedStepDiscrete');
set_param(modelName,'SupportContinuousTime','on');
set_param(modelName,'LaunchReport','off');
set_param(modelName,'InitFltsAndDblsToZero','on');

% Generate code
slbuild(modelName);

% Create Bug Finder project configuration
psprjCfgBF = polyspace.ModelLinkOptions(modelName);
% Enable coding rules, such as MISRA C:2012
psprjCfgBF.CodingRulesCodeMetrics.EnableMisraC3 = true;
psprjCfgBF.CodingRulesCodeMetrics.MisraC3Subset = 'all';
% Deactivate Bug Finder defects
psprjcfg.BugFinderAnalysis.EnableChecker = false;

% Specify results folder for bugfinder analysis
psprjCfgBF.ResultsDir = 'BF_newResfolder';

% Associate the project configurations with a Polyspace project
bfProj = polyspace.Project;
bfProj.Configuration = psprjCfgBF;

% Set verification mode as Bug Finder
bfStatus = bfProj.run('bugfinder');
% obtain BF results in a table
BF_results = bfProj.Results.getResults('readable');
```

```
% Create a new project for Code Prover verification
cpProj = polyspace.Project;

% Create a new configuration object for Code Prover verification
psprjCfgCP =  polyspace.ModelLinkOptions(modelName);
psprjCfgCP.CodingRulesCodeMetrics.EnableMisraC3 = false;

% Specify results folder for Code Prover analysis
psprjCfgCP.ResultsDir = 'CP_newResfolder';

% Associate the project configurations with a Polyspace project
cpProj.Configuration = psprjCfgCP;

% Set verification mode as Code Prover
cpStatus = cpProj.run('codeprover');
% obtain CP results in a table
CP_results = cpProj.Results.getResults('readable');
```

Produce a Polyspace Report Containing Run-Time Errors, Coding Rule Violations, and Code Metrics

Previously, you configured a Code Prover verification to produce a single report containing run-time errors, code metrics, coding rule violations, and other results. To migrate to using Bug Finder, configure the same options and run separate Bug Finder and Code Prover analyses. See “Compute Code Metrics, Check for Run-Time Errors and Coding Rule Violations”.

The Bug Finder and Code Prover results are summarized in separated reports.

To produce a combined report containing Bug Finder and Code Prover results, use `polyspace-report-generator`. For instance, if your Bug Finder and Code Prover results are saved in the folders `BF_results` and `CP_results`, use this command at the command prompt:

```
polyspace-report-generator ^
-template %template_path% ^
-results-dir "CP_Results","BF_Results"
```

Sample Batch Script

```
@echo off
Rem Specify the path for source
set source=^
"C:\Program Files\Polyspace\R2021b\polyspace\examples\cxx\^
Bug_Finder_Example\sources\numerical.c"
Rem Using Developer.rpt as template
set template_path=^
"C:\Program Files\Polyspace\R2021b\toolbox\polyspace\psrptgen\^
templates\Developer.rpt"
Rem making results directory for Bug Finder and Code Prover Run
mkdir bfResults;
mkdir cpResults;
Rem Start Bug Finder analysis
polyspace-bug-finder -sources %source%^
 -results-dir "%CD%\bfResults"^
 -code-metrics ^
 -misra3 all-rules ^
 -lang c ^
```

```
 -checkers none
 Rem Start Code Prover analysis
polyspace-code-prover -sources %source%^
 -results-dir "%CD%\cpResults"^
 -lang c ^
 -main-generator
Rem Start Report generation
polyspace-report-generator ^
-template %template_path% ^
-results-dir "%CD%\cpResults","%CD%\bfResults"
```

Check for Protected and Unprotected Shared Global Variables

Previously, to perform this task, you specified the entry point functions and temporally exclusive functions in your code, and then computed code metrics by using Code Prover. The recommended tool for performing this task is to use the global variable checks in Code Prover instead.

- In the Polyspace user interface, configure the same options you did before, and then click **Run Code Prover**. You do not need to check **Calculate Code Metrics**.
- At the command line, run a Code Prover verification by using the same analysis options that you used before. Omit `-code-metrics`.

After the verification completes, in the Results List, the protected shared global variables are flagged by green checks, and the potentially unprotected shared global variables are flagged by orange checks.

Family	Information	File	Function	Status
Run-time Check		4 6		
Orange Check		4		
Green Check		6		
Global Variable		1 1		
Shared		1 1		
Potentially unprotected variable		1		
?	Variable: shared_var	shared__global.c	_init_globals()	Unreviewed
Protected variable		1		
✓	Variable: Count	shared__global.c	_init_globals()	Unreviewed

See “Global Variables”.

Calculate Stack Usage

Previously, to calculate stack usage, you configured your Code Prover analysis in the Polyspace UI to compute code metrics or specified the option `-code-metrics`. The recommended tool for performing this task is to use the analysis option `Calculate stack usage (-stack-usage)`.

- In the Polyspace user interface, check **Calculate stack usage** in the **Check Behavior** pane, and then click **Run Code Prover**. You do not need to check **Calculate Code Metrics**.
- At the command line, run a Code Prover verification by using the same analysis options that you used before. Use `-stack-usage` instead of `-code-metrics`.

After the verification completes, in the Results List, the stack usage metrics are listed.

See Also

More About

- "Justify Coding Rule Violations Using Code Prover Checks" on page 31-9
- "Check for and Review Coding Standard Violations"
- "AUTOSAR C++14 Rules"
- "CERT C Rules and Recommendations"
- "CERT C++ Rules"
- "Polyspace Support for Coding Standards"
- "Justify Coding Rule Violations Using Code Prover Checks"
- "Coding Standards & Code Metrics"

Polyspace Coverage of Coding Standards

- “ Polyspace Support for Coding Standards” on page 17-2
- “MISRA C:2004 and MISRA AC AGC Coding Rules” on page 17-9
- “Required or Mandatory MISRA C:2012 Rules Supported by Polyspace Bug Finder” on page 17-43
- “Decidable MISRA C:2012 Rules Supported by Polyspace Bug Finder” on page 17-54
- “Undecidable MISRA C:2012 Rules and Directives Supported by Polyspace Bug Finder” on page 17-64
- “Essential Types in MISRA C:2012 Rules 10.x” on page 17-69
- “Unsupported MISRA C:2012 Guidelines” on page 17-71
- “Required and Statically Enforceable CERT C Rules Supported by Polyspace Bug Finder” on page 17-72
- “Required MISRA C++:2008 Coding Rules Supported by Polyspace Bug Finder” on page 17-80
- “JSF AV C++ Coding Rules” on page 17-97
- “Required AUTOSAR C++14 Coding Rules Supported by Polyspace Bug Finder” on page 17-122
- “Statically Enforceable AUTOSAR C++14 Rules Supported by Polyspace Bug Finder” on page 17-153

Polyspace Support for Coding Standards

Polyspace Bug Finder and Polyspace as You Code support various coding standards. Check the compliance of your code with these standards by analyzing your individual translation units in the IDE, and then analyzing your entire source code during integration. Polyspace as You Code supports a subset of rules that Bug Finder supports. See “Checkers Deactivated in Polyspace as You Code Analysis”.

Summary of Polyspace Support

Standard	Release	Statically Enforceable Rules	Required or Mandatory Rules
“AUTOSAR C++14”	10-31-2018	349 out of 349 rules in the standard	337 out of 362 rules in the standard
“MISRA C++:2008”	June 2008	[a]	195 out of 198 rules in the standard
“MISRA C:2012”	• March 2013 • April 2016 (Amendment 1) • June 2017 (TC1) • January 2018 (Amendment 2)	122 out of 122 rules in the standard	126 out of 126 rules in the standard
“CERT C”	2016	120 out of 120 rules in the standard	120 out of 120 rules in the standard

a MISRA C++:2008 standard does not categorize rules based on their static enforceability

Coding standards categorize the rules based on their obligation level or their static enforceability. Polyspace supports rules that are considered nonenforceable or partially enforceable by the standards. Enforcing these rules require a manual review process, which can be assisted by the Polyspace results.

AUTOSAR C++14

The AUTOSAR C++14 standard categorizes the rules based on their obligation level and enforcement by static analysis.

Obligation Level

Category	Rules Implemented in Bug Finder	Rules in the Standard
Required: The code must follow these rules.	337	362
Advisory: The code is advised to follow these rules to a reasonable practical extent.	33	35
	Total: 370	

Enforcement by Static Analysis Tool

Category	Rules Implemented in Bug Finder	Rules in the Standard
Automated: Static analysis tools can detect all violation of these rules.	327	327[a]
Partially automated: Static analysis tools cannot detect all possible violations of these rules. You need manual code review or other tools to completely enforce these rules. Polyspace shows the subset of all possible issues. For details about which issues Polyspace detects for a particular rule, see the **Polyspace Implementation** section in the reference page of the rule.	22	22
Nonautomated: Static analysis tools cannot detect all possible violations of these rules. You need manual code review or other tools to completely enforce these rules. Polyspace shows the subset of all possible issues. For details about which issues Polyspace detects for a particular rule, see the **Polyspace Implementation** section in the reference page of the rule.	21	46

a The AUTOSAR C++14 standard contains 329 **Automated** rules. The rules A0-4-3 and A1-4-3 are not enforceable by a static analysis tool. These rules might be enforced by a compiler.

The **Automated** and **Partially automated** rules are statically enforceable. In total, Polyspace supports 349 statically enforceable rules and 337 required rules. See:

- “Statically Enforceable AUTOSAR C++14 Rules Supported by Polyspace Bug Finder”.
- “Required AUTOSAR C++14 Coding Rules Supported by Polyspace Bug Finder”.

MISRA C++:2008

The MISRA C++:2008 standard categorizes the rules based on their obligation level.

Category	Rules Implemented in Bug Finder	Rules in the Standard
Required: The code must follow these rules.	195	198

Category	Rules Implemented in Bug Finder	Rules in the Standard
Advisory: The code is advised to follow these rules to a reasonable practical extent.	18	18
Document: These rules are associated with different features including #pragmas, floating-point arithmetic, or bit fields. Whenever these features are used, the code must follow the associated rule.	1	12
	Total: 214	

In total Polyspace supports 193 out of 198 required MISRA C++:2008 rules. See “Required MISRA C++:2008 Coding Rules Supported by Polyspace Bug Finder”.

MISRA C:2012

The MISRA C:2012 standard classifies the guidelines as either a rule or a directive. Polyspace supports the original MISRA C:2012 standard as well as the technical corrigendum 1, amendments 1, and 2. See “Polyspace Support for MISRA C: 2012 Amendments”.

MISRA C:2012 Rules

A rule is a guideline that can be described completely. Compliance with a rule can be checked statically with some limitation. The rules are further categorized based on different properties.

Obligation Level

Category	Rules Implemented in Bug Finder	Rules in the Standard
Mandatory: These are guidelines that compliant C code must follow. The standard does not permit deviations from these guidelines.	16	16
Required: These are guidelines that compliant C code must follow. The standard permits only the deviations that you formally record and authorize.	110	110
Advisory: These are recommended guidelines. The standard permits deviation from these guidelines without any formal record. It is a best practice to follow these guidelines to a reasonably practical degree and record the deviations.	32	32
	Total: 158	

Compliant C code must follow the **Mandatory** and **Required** rules. Polyspace supports all such rules.

Static Enforceability

Category	Rules Implemented in Bug Finder	Rules in the Standard
Decidable: A rule is decidable if a static analysis tool can check compliance with the rule in every possible case.	122	122
Undecidable: A rule is undecidable if a static analysis tool can check compliance to it only in certain cases. Polyspace shows the subset of all possible issues. For details about which issues Polyspace detects for a particular rule, see the **Polyspace Implementation** section in the reference page of the rule.	36	36

Analysis Scope

Category	Rules Implemented in Bug Finder	Rules in the Standard
Single Translation Unit: You can find all violations of these rules by checking each translation unit of a project individually.	109	109
System: You can find all violations of these rules only by analyzing the entire project or system.	49	49

Polyspace supports 122 out of 122 decidable MISRA C:2012 rules. See "Decidable MISRA C:2012 Rules Supported by Polyspace Bug Finder".

MISRA C:2012 Directives

Directives are guidelines that cannot be completely described. Checking compliance with these directives requires more information in addition to the code. Static analysis might assist in checking compliance with directives. The directives are categorized based on obligation level.

Obligation Level

Category	Directives Implemented in Bug Finder	Directives in the Standard
Required: These are guidelines that compliant C code must follow. The standard permits only the deviations that you formally record and authorize.	9	10
Advisory: These are recommended guidelines. The standard permits deviation from these guidelines without any formal record. It is a best practice to follow these guidelines to a reasonably practical degree and record the deviations.	6	7

Static Enforceability

Category	Directive Implemented in Bug Finder	Directive in the Standard
Decidable: A directive is decidable if a static analysis tool can check compliance with the directive in every possible case.	0	0
Undecidable: A directive is undecidable if a static analysis tool can check compliance to it only in certain cases. Polyspace shows the subset of all possible issues. For details about which issues Polyspace detects for a particular directive, see the **Polyspace Implementation** section in the reference page of the directive.	15	17

Polyspace supports 36 out of 36 undecidable rules as well as 15 undecidable directives. See "Undecidable MISRA C:2012 Rules and Directives Supported by Polyspace Bug Finder".

CERT C

Polyspace supports all statically enforceable rules in the CERT C standard. The standard categorizes the guidelines into rules and recommendations. Polyspace does not support rules that are being removed or under construction.

Category	Checks Implemented in Bug Finder	Checks in the Standard
Rule: These guidelines are required. Violation of these guidelines might compromise the safety, security, or reliability of a system. Static analysis tools can enforce compliance with these guidelines.	120	120
Recommendation: These guidelines are meant to improve the readability, safety, and security of a system. Static analysis can only detect a subset of violations of these guidelines. Polyspace shows the subset of all possible issues. For details about which issues Polyspace detects, see the reference page of these rules.	83	183

Other

Polyspace also supports these coding rule standards.

Standard	Rules Implemented in Bug Finder
MISRA C:2004	132 out of 142 rules in the standard
MISRA AC AGC	130 out of 142 in the standard
ISO/IEC TS 17961	46 out of 46 in the standard
JSF AV C++	160 out of 234 in the standard
CERT C++	153 out of 163 in the standard
CWE	102 (version 4.9), including: • 72 out of 82 C specific rules (CWE-658). • 75 out of 86 C++ specific rules (CWE-659).

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)` | `Check MISRA C++:2008 (-misra-cpp)` | `Check SEI CERT-C++ (-cert-cpp)` | `Check MISRA C:2012 (-misra3)` | `Check MISRA C:2004 (-misra2)` | `Check SEI CERT-C (-cert-c)` | `Check CWE (-cwe)`

More About

- “Checkers Deactivated in Polyspace as You Code Analysis”
- “Polyspace Support for MISRA C: 2012 Amendments”
- “Decidable MISRA C:2012 Rules Supported by Polyspace Bug Finder”
- “Undecidable MISRA C:2012 Rules and Directives Supported by Polyspace Bug Finder”
- “Required AUTOSAR C++14 Coding Rules Supported by Polyspace Bug Finder”
- “Statically Enforceable AUTOSAR C++14 Rules Supported by Polyspace Bug Finder”
- “Required MISRA C++:2008 Coding Rules Supported by Polyspace Bug Finder”
- “Required and Statically Enforceable CERT C Rules Supported by Polyspace Bug Finder”

MISRA C:2004 and MISRA AC AGC Coding Rules

In this section...

"Supported MISRA C:2004 and MISRA AC AGC Rules" on page 17-9
"Troubleshooting" on page 17-9
"List of Supported Coding Rules" on page 17-9
"Unsupported MISRA C:2004 and MISRA AC AGC Rules" on page 17-41

Note Starting in a future release, Code Prover will not support checking compliance with external coding standards and calculating code metrics. Migrate to Bug Finder for these workflows. See "MISRA C:2004 and MISRA AC AGC Coding Rules".

Supported MISRA C:2004 and MISRA AC AGC Rules

The following tables list MISRA C:2004 coding rules that the Polyspace coding rules checker supports. Details regarding how the software checks individual rules and any limitations on the scope of checking are described in the "Polyspace Specification" column.

Note The Polyspace coding rules checker:

- Supports MISRA-C:2004 Technical Corrigendum 1 for rules 4.1, 5.1, 5.3, 6.1, 6.3, 7.1, 9.2, 10.5, 12.6, 13.5, and 15.0.
- Checks rules specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

The software reports most violations during the compile phase of an analysis. However, the software detects violations of rules 9.1 (`Non-initialized variable`), 12.11 (one of the overflow checks) using `-scalar-overflows-checks signed-and-unsigned`), 13.7 (dead code), 14.1 (dead code), 16.2 and 21.1 during code analysis, and reports these violations as run-time errors.

Note Some violations of rules 13.7 and 14.1 are reported during the compile phase of analysis.

Troubleshooting

If you expect a rule violation but do not see it, check out "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

List of Supported Coding Rules

- "Environment" on page 17-11
- "Language Extensions" on page 17-12
- "Documentation" on page 17-15

- “Character Sets” on page 17-15
- “Identifiers” on page 17-15
- “Types” on page 17-16
- “Constants” on page 17-17
- “Declarations and Definitions” on page 17-17
- “Initialisation” on page 17-20
- “Arithmetic Type Conversion” on page 17-21
- “Pointer Type Conversion” on page 17-24
- “Expressions” on page 17-25
- “Control Statement Expressions” on page 17-27
- “Control Flow” on page 17-29
- “Switch Statements” on page 17-31
- “Functions” on page 17-32
- “Pointers and Arrays” on page 17-33
- “Structures and Unions” on page 17-34
- “Preprocessing Directives” on page 17-34
- “Standard Libraries” on page 17-37
- “Runtime Failures” on page 17-41

Environment

N.	MISRA Definition	Messages in report file	Polyspace Implementation
1.1	All code shall conform to ISO 9899:1990 “Programming languages - C”, amended and corrected by ISO/IEC 9899/ COR1:1995, ISO/IEC 9899/ AMD1:1995, and ISO/IEC 9899/ COR2:1996.	The text *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* precedes each of the following messages: • 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.)		The text *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* precedes each of the following messages: • 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.	No warnings if code is encapsulated in the following: • `asm` functions or `asm pragma` • Macros

N.	MISRA Definition	Messages in report file	Polyspace Implementation
2.2	Source code shall only use /* */ style comments	C++ comments shall not be used.	C++ comments are handled as comments but lead to a violation of this MISRA rule **Note**: This rule cannot be annotated in the source code.
2.3	The character sequence /* shall not be used within a comment	The character sequence /* shall not appear within a comment.	This rule violation is also raised when the character sequence /* inside a C++ comment. **Note**: This rule cannot be annotated in the source code.

N.	MISRA Definition	Messages in report file	Polyspace Implementation

2.4	Sections of code should not be "commented out"	Sections of code should not be "commented out"	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.
			
			The checker does not flag the following comments even if they contain code:
			
			• Doxygen comments beginning with `/**` or `/*!`.
			• Comments that repeat the same symbol several times, for instance, the symbol = here:
			
			`/* ==========`
			` * A comment`
			` * ==========*/`
			
			• Comments on the first line of a file.
			• Comments that mix the C style (`/* */`) and C++ style (`//`).
			
			The checker considers that these comments are meant for documentation purposes or entered deliberately with some forethought.

Documentation

Rule	MISRA Definition	Messages in report file	Polyspace Implementation
3.4	All uses of the #*pragma* directive shall be documented and explained.	All uses of the #pragma 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 `Allowed pragmas (-allowed-pragmas)`. 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.	\<character> 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. *This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”*.
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.	• 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 *This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*
5.5	No object or function identifier with a static storage duration should be reused.	{static identifier/parameter name}'%s' should not be reused. (already used as {static identifier/parameter name} with static storage duration at %s:%d)	Warning when a static name is reused as another identifier name
5.6	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.	{member name}'%s' should not be reused. (already used as {member name} at %s:%d)	Warning when an `idf` in a namespace is reused in another namespace *This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*
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: • Different functions have parameters with the same name • Different functions have local variables with the same name • A function has a local variable that has the same name as a parameter of another function

Types

N.	MISRA Definition	Messages in report file	Polyspace Implementation
6.1	The plain char type shall be used only for the storage and use of character values	Only permissible operators on plain chars are '=', '==' or '!=' operators, explicit casts to integral types and '?' (for the 2nd and 3rd operands)	Warning when a plain char is used with an operator other than =, ==, !=, explicit casts to integral types, or as the second or third operands of the ? operator.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
6.2	Signed and unsigned char type shall be used only for the storage and use of numeric values.	• 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	*typedefs* that indicate size and signedness should be used in place of the basic types	typedefs 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 *unsigned int* or *signed int*.	Bit fields shall only be defined to be of type unsigned int or signed int.	
6.5	Bit fields of type *signed int* 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.	• 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.	• Function XX has no complete prototype visible at call. • Function XX has no prototype visible at definition.	Prototype visible at call must be complete.
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated	Whenever an object or function is declared or defined, its type shall be explicitly stated.	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
8.3	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.	Definition of function 'XX' incompatible with its declaration.	Assumes that rule 8.1 is not violated. The rule is restricted to compatible types. Can be turned to Off
8.4	If objects or functions are declared more than once their types shall be compatible.	• If objects or functions are declared more than once their types shall be compatible. • Global declaration of 'XX' function has incompatible type with its definition. • Global declaration of 'XX' variable has incompatible type with its definition.	Violations of this rule might be generated during the link phase. *This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis"*.
8.5	There shall be no definitions of objects or functions in a header file	• Object 'XX' should not be defined in a header file. • Function 'XX' should not be defined in a header file. • Fragment of function should not be defined in a header file.	Tentative definitions are considered as definitions. For objects with file scope, tentative definitions are declarations that: • Do not have initializers. • Do not have storage class specifiers, or have the `static` 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** `addrescape`.
8.7	Objects shall be defined at block scope if they are only accessed from within a single function	Object 'XX' should be declared at block scope.	Restricted to static objects.
8.8	An external object or function shall be declared in one file and only one file	Function/Object 'XX' has external declarations in multiple files.	Restricted to explicit extern declarations (tentative definitions are ignored). Polyspace considers that variables or functions declared `extern` in a non-header file violate this rule.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
8.9	An identifier with external linkage shall have exactly one external definition.	• Procedure/Global variable XX multiply defined. • Forbidden multiple tentative definitions for object XX • Global variable has multiple tentative definitions • Undefined global variable XX	The checker flags multiple definitions only if the definitions occur in different files. No warnings appear on predefined symbols. *This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”*.
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.	Assumes that 8.1 is not violated. No warning if 0 uses. If your code does not contain a `main` function and you use options such as `Variables to initialize (-main-generator-writes-variables)` with value `custom` to explicitly specify a set of variables to initialize, the checker does not flag those variables. The checker assumes that in a real application, the file containing the `main` must initialize the variables in addition to any file that currently uses them. Therefore, the variables must be used in more than one translation unit. *This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”*.
8.11	The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage	static storage class specifier should be used on internal linkage symbol XX.	
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization	Size of array 'XX' should be explicitly stated.	

Initialisation

N.	MISRA Definition	Messages in report file	Polyspace Implementation
9.1	All automatic variables shall have been assigned a value before being used.		Polyspace reports a violation of this rule if your code contains these issues: • `Non-initialized variable` • `Non-initialized pointer`
9.2	Braces shall be used to indicate and match the structure in the nonzero initialisation of arrays and structures.	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.	
9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	

Arithmetic Type Conversion

N.	MISRA Definition	Messages in report file	Polyspace Implementation
10.1	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: • 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	• 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.	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. An expression of bool or enum types has int as underlying type. Plain char may have signed or unsigned underlying type (depending on Polyspace 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 is not token into account and it assumes that only signed | unsigned int are used for bitfield (Rule 6.4). This rule violation is not produced on operations involving pointers. No violation reported when: • The implicit conversion is a type widening, without change of signedness of integer • The expression is an argument expression or a return expression No violation reported when the following are true: • 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

<table>
<tr><th>N.</th><th>MISRA Definition</th><th>Messages in report file</th><th>Polyspace Implementation</th></tr>
<tr><td></td><td></td><td></td><td>the constant value or the result of the operation.
• The expression is an argument expression or a return expression or an operand expression of a non-bitwise operator.

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.</td></tr>
<tr><td>10.2</td><td>The value of an expression of floating type shall not be implicitly converted to a different type if
• 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</td><td>• 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.</td><td>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.

No violation reported when:
• The implicit conversion is a type widening
• The expression is an argument expression or a return expression.</td></tr>
</table>

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.	• 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 `i` but not the second. In the first assignment, a composite expression `i+1` 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.
			
			`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 */`
			• 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 token 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 *unsigned char* or *unsigned short*, 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 *unsigned* types	No explicit 'U suffix on constants of an unsigned type.	Warning when the type determined from the value and the base (octal, decimal or hexadecimal) is unsigned and there is no suffix `u` or `U`. For example, when the size of the `int` and `long int` data types is 32 bits, the coding rule checker will report a violation of rule 10.6 for the following line: `int a = 2147483648;` There is a difference between decimal and hexadecimal constants when `int` and `long int` are not the same size.

Pointer Type Conversion

N.	MISRA Definition	Messages in report file	Polyspace Implementation
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type	Conversion shall not be performed between a pointer to a function and any type other than an integral type.	Casts and implicit conversions involving a function pointer. Casts or implicit conversions from `NULL` or `(void*)0` do not give any warning.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.	There is also a warning on qualifier loss This rule maps to **ISO/IEC TS 17961 ID** `alignconv`.

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 *const* or *volatile* qualification from the type addressed by a pointer	A cast shall not be performed that removes any *const* or *volatile* 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.	• The value of `'sym'` depends on the order of evaluation. • The value of volatile `'sym'` depends on the order of evaluation because of multiple accesses.	Rule 12.2 check assumes that no assignment in expressions that yield a Boolean values (rule 13.1). The expression is a simple expression of symbols. `i = i++;` is a violation, but `tab[2] = tab[2]++;` is not a violation.
12.3	The `sizeof` operator should not be used on expressions that contain side effects.	The `sizeof` 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 && or || operator shall not contain side effects.	The right hand operand of a logical && or || operator shall not contain side effects.	No warning on volatile accesses
12.5	The operands of a logical && or || shall be primary-expressions.	• operand of logical && is not a primary expression • operand of logical || is not a primary expression • The operands of a logical && or || shall be primary-expressions.	During preprocessing, violations of this rule are detected on the expressions in #if directives. Allowed exception on associatively (a && b && c), (a || b || c).

<table>
<tr><th>N.</th><th>MISRA Definition</th><th>Messages in report file</th><th>Polyspace Implementation</th></tr>
<tr><td>12.6</td><td>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 !).</td><td>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 '?:'.</td><td>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.

Some operators may return Boolean-like expressions, for example, <code>(var == 0)</code>.

Consider the following code:

<code>unsigned char flag;</code>
<code>if (!flag)</code>

The rule checker reports a violation of rule 12.6:

<code>Operand of '!' logical operator should be effectively Boolean.</code>

The operand <code>flag</code> is not a Boolean but an <code>unsigned char</code>.

To be compliant with rule 12.6, the code must be rewritten either as

<code>if (!(flag != 0))</code>

or

<code>if (flag == 0)</code>

The use of the option <code>-boolean-types</code> may increase or decrease the number of warnings generated.</td></tr>
<tr><td>12.7</td><td>Bitwise operators shall not be applied to operands whose underlying type is signed</td><td>[~/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</td><td>The underlying type for an integer is signed when: it does not have a u or U suffix it is small enough to fit into a 64 bits signed number</td></tr>
</table>

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.	• 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)..	The numbers that are manipulated in preprocessing directives are 64 bits wide so that valid shift range is between 0 and 63 Check is also extended onto bitfields with the field width or the width of the base type when it is within a complex expression
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	• Unary - on operand of unsigned underlying type XX. • Minus operator applied to an expression whose underlying type is unsigned	The underlying type for an integer is signed when: • 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.	Warning when:
			• 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:
			`union {`
			` float f;`
			` int i;`
			`} …`

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 `-boolean-types` 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 directs tests only.
13.4	The controlling expression of a *for* statement shall not contain any objects of floating type	The controlling expression of a for statement shall not contain any objects of floating type	If *for* index is a variable symbol, checked that it is not a float.
13.5	The three expressions of a *for* statement shall be concerned only with loop control	• 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 for loops are allowed: (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 for 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 *for* loop for iteration counting should not be modified in the body of the loop.	Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop.	Detect only direct assignments if the for 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	• 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.
			During compilation, the checker covers comparisons with at least one constant operand. Some violations of this rule are reported through the `Dead code` and `Useless if` checkers.
			
			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:
			
			`enum ec {RED, BLUE, GREEN} col;`
			`for(col=RED; col<=GREEN; col++)`
			`{}`
			
			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:
			
			`enum ec {RED, BLUE, GREEN} col;`
			`for(col=RED; (int)col<=GREEN; 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.	
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: • have at least one side effect however executed, or • cause control flow to change	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
14.3	Before preprocessing, a null statement shall occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.	A null statement shall appear on a line by itself	We assume that a ';' is a null statement when it is the first character on a line (excluding comments). The rule is violated when: • 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 *goto* statement shall not be used.	The goto statement shall not be used.	
14.5	The *continue* statement shall not be used.	The continue statement shall not be used.	
14.6	For any iteration statement there shall be at most one *break* statement used for loop termination	For any iteration statement there shall be at most one break statement used for loop termination	
14.7	A function shall have a single point of exit at the end of the function	A function shall have a single point of exit at the end of the function	
14.8	The statement forming the body of a *switch, while, do while* or *for* statement shall be a compound statement	• 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 *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	• 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 *if else if* constructs should contain a final *else* 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.	Warning on declarations or any statements before the first switch case.
			
			Warning on label or jump statements in the body of switch cases.
			
			On the following example, the rule is displayed in the log file at line 3:
			
			`1 ...`
			`2 switch(index) {`
			`3  var = var + 1;`
			`// RULE 15.0`
			`// violated`
			`4case 1: ...`
			
			The code between switch statement and first case is checked as dead code by Polyspace. It follows ANSI standard behavior.
			
			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.

15.1	A switch label shall only be used when the most closely-enclosing compound statement is the body of a *switch* 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 *break* 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 *switch* statement shall be the *default* clause	The final clause of a switch statement shall be the default clause	
15.4	A *switch* 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 `-boolean-types` may increase the number of warnings generated.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
15.5	Every *switch* statement shall have at least one *case* clause	Every switch statement shall have at least one case clause	

Functions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
16.1	Functions shall not be defined with variable numbers of arguments.	Function XX should not be defined as varargs.	
16.2	Functions shall not call themselves, either directly or indirectly.	Function %s should not call itself.	The checker reports each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported. You can calculate the total number of recursion cycles using the code complexity metric `Number of Recursions`.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.	Identifiers shall be given for all of the parameters in a function prototype declaration.	Assumes Rule **8.6** is not violated.
16.4	The identifiers used in the declaration and definition of a function shall be identical.	The identifiers used in the declaration and definition of a function shall be identical.	Assumes that rules **8.8**, **8.1** and **16.3** are not violated. All occurrences are detected.
16.5	Functions with no parameters shall be declared with parameter type *void*.	Functions with no parameters shall be declared with parameter type void.	Definitions are also checked.
16.6	The number of arguments passed to a function shall match the number of parameters.	• 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 pointer to `const` if the pointer is not used to modify the addressed object.	Pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.	Warning if a non-`const` 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 `const` 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.	The checker flags functions with non-`void` return if the return value is not used or not explicitly cast to a `void` type. The checker does not flag the functions `memcpy`, `memset`, `memmove`, `strcpy`, `strncpy`, `strcat`, `strncat` because these functions simply return a pointer to their first arguments.

Pointers and Arrays

N.	MISRA Definition	Messages in report file	Polyspace Implementation
17.1	Pointer arithmetic shall only be applied to pointers that address an array or array element.	Pointer arithmetic shall only be applied to pointers that address an array or array element.	
17.2	Pointer subtraction shall only be applied to pointers that address elements of the same array	Pointer subtraction shall only be applied to pointers that address elements of the same array.	Polyspace reports a violation when you subtract that are null or that point to elements in different arrays.
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	Polyspace reports a violation when you compare pointers that are null or that point to elements in different arrays. The relational operators for the comparison are >, <, >=, and <=.
17.4	Array indexing shall be the only allowed form of pointer arithmetic.	Array indexing shall be the only allowed form of pointer arithmetic.	Warning on: • Operations on pointers. (`p+I`, `I+p`, and `p-I`, where `p` is a pointer and `I` an integer). • Array indexing on nonarray pointers.
17.5	A type should not contain more than 2 levels of pointer indirection	A type should not contain more than 2 levels of pointer indirection	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.	Pointer to a parameter is an illegal return value. Pointer to a local is an illegal return value.	Warning when assigning address to a global variable, returning a local variable address, or returning a parameter address. This rule maps to **ISO/IEC TS 17961 ID** `accfree`.

Structures and Unions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
18.1	All structure or union types shall be complete at the end of a translation unit.	All structure or union types shall be complete at the end of a translation unit.	Warning for all incomplete declarations of structs or unions.
18.2	An object shall not be assigned to an overlapping object.	• 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	#include statements in a file shall only be preceded by other preprocessors directives or comments	#include statements in a file shall only be preceded by other preprocessors directives or comments	A message is displayed when a #include 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 #include directives	• A message is displayed on characters ', " or /* between < and > in #include <filename> • A message is displayed on characters ', or /* between " and " in #include "filename"	
19.3	The *#include* directive shall be followed by either a <filename> or "filename" sequence.	• '#include' expects "FILENAME" or <FILENAME> • '#include_next' expects "FILENAME" or <FILENAME>	

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.	We assume that a macro definition does not violate this rule when it expands to: • 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 #undefd within a block.	• 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	• 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.	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`. 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,`.
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, #ifndef <control macro> #define <control macro> <contents> #endif or, #ifndef <control macro> #error ... #else #define <control macro> <contents> #endif 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 #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	• '#elif' not within a conditional. • '#else' not within a conditional. • '#elif' not within a conditional. • '#endif' not within a conditional. • unbalanced '#endif'. • unterminated '#if' conditional. • unterminated '#ifdef' conditional. • unterminated '#ifndef' 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.	• The macro '<name> shall not be redefined. • The macro '<name> shall not be undefined.	

<table>
<tr><th>N.</th><th>MISRA Definition</th><th>Messages in report file</th><th>Polyspace Implementation</th></tr>
<tr><td>20.2</td><td>The names of standard library macros, objects and functions shall not be reused.</td><td>Identifier XX should not be used.</td><td>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.

Tentative definitions are considered as definitions. For objects with file scope, tentative definitions are declarations that:
• Do not have initializers.
• Do not have storage class specifiers, or have the <code>static</code> specifier</td></tr>
</table>

N.	MISRA Definition	Messages in report file	Polyspace Implementation
20.3	The validity of values passed to library functions shall be checked.	Validity of values passed to library functions shall be checked	Warning for argument in library function call if the following are all true: • 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. The library function can be one of the following : `sqrt`, `tan`, `pow`, `log`, `log10`, `fmod`, `acos`, `asin`, `acosh`, `atanh`, or `atan2`. You might be using a custom library of mathematical functions. If a custom library function have the same domain and range as another function from the standard library, you can extend this checker to check the custom library function. See “Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries”. A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”. By default, a Bug Finder analysis does not recognize infinities and `NaNs`. Operations that results in infinities and `NaNs` might be flagged as defects. To handle infinities and `NaN` values in your code, use the option `Consider`

N.	MISRA Definition	Messages in report file	Polyspace Implementation
			`non finite floats (-allow-non-finite-floats)`.
20.4	Dynamic heap memory allocation shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the dynamic heap memory allocation functions are actually macros and the macro is expanded in the code, this rule is detected as violated. Assumes rule **20.2** is not violated.
20.5	The error indicator errno shall not be used	The error indicator errno shall not be used	Assumes that rule **20.2** is not violated
20.6	The macro *offsetof*, in library <stddef.h>, shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	Assumes that rule **20.2** is not violated
20.7	The *setjmp* macro and the *longjmp* function shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the longjmp function is actually a macro and the macro is expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.8	The signal handling facilities of <signal.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case some of the signal functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.9	The input/output library <stdio.h> shall not be used in production code.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the input/output library functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.10	The library functions atof, atoi and atoll from library <stdlib.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the atof, atoi and atoll functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.11	The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the abort, exit, getenv and system functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated

N.	MISRA Definition	Messages in report file	Polyspace Implementation
20.12	The time handling functions of library <time.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the time handling functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated

Runtime Failures

N.	MISRA Definition	Messages in report file	Polyspace Implementation
21.1	Minimization of runtime failures shall be ensured by the use of at least one of: • static verification tools/ techniques; • dynamic verification tools/ techniques; • explicit coding of checks to handle runtime faults.		

Unsupported MISRA C:2004 and MISRA AC AGC Rules

The Polyspace coding rules checker does not check the following MISRA C:2004 coding rules. These rules cannot be enforced because they are outside the scope of Polyspace software. They may concern documentation, dynamic aspects, or functional aspects of MISRA rules. The **Additional Information** column describes the reason each rule is not checked.

Environment

Rule	Description	Additional Information
1.2 (Required)	No reliance shall be placed on undefined or unspecified behavior	Not statically checkable unless the data dynamic properties is taken into account
1.3 (Required)	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the language/compilers/assemblers conform.	It is a process rule method.
1.4 (Required)	The compiler/linker/Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.	To observe this rule, check your compiler documentation.
1.5 (Advisory)	Floating point implementations should comply with a defined floating point standard.	To observe this rule, check your compiler documentation.

Documentation

Rule	Description	Additional Information
3.1 (Required)	All usage of implementation-defined behavior shall be documented.	To observe this rule, check your compiler documentation. Error detection is based on undefined behavior, according to choices made for implementation- defined constructions.
3.2 (Required)	The character set and the corresponding encoding shall be documented.	To observe this rule, check your compiler documentation.
3.3 (Advisory)	The implementation of integer division in the chosen compiler should be determined, documented and taken into account.	To observe this rule, check your compiler documentation.
3.5 (Required)	The implementation-defined behavior and packing of bitfields shall be documented if being relied upon.	To observe this rule, check your compiler documentation.
3.6 (Required)	All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.	To observe this rule, check your compiler documentation.

Structures and Unions

Rule	Description	Additional Information
18.3 (Required)	An area of memory shall not be reused for unrelated purposes.	"purpose" is functional design issue.

Required or Mandatory MISRA C:2012 Rules Supported by Polyspace Bug Finder

The MISRA C:2012 standard classifies the rules that compliant C code must follow as **Required** and **Mandatory**. In total, Polyspace supports 126 out of 126 such rules.

Mandatory Rules

Compliant C code must follow these coding rules. The standard does not permit deviation from these rules. Polyspace supports 16 out of 16 such rules.

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 12.5	The `sizeof` operator shall not have an operand which is a function parameter declared as “array of type”	`MISRA C:2012 Rule 12.5`
MISRA C:2012 Rule 13.6	The operand of the sizeof operator shall not contain any expression which has potential side effects	`MISRA C:2012 Rule 13.6`
MISRA C:2012 Rule 17.3	A function shall not be declared implicitly	`MISRA C:2012 Rule 17.3`
MISRA C:2012 Rule 17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression	`MISRA C:2012 Rule 17.4`
MISRA C:2012 Rule 17.6	The declaration of an array parameter shall not contain the static keyword between the [ ]	`MISRA C:2012 Rule 17.6`
MISRA C:2012 Rule 19.1	An object shall not be assigned or copied to an overlapping object	`MISRA C:2012 Rule 19.1`
MISRA C:2012 Rule 21.13	Any value passed to a function in `<ctype.h>` shall be representable as an `unsigned char` or be the value EOF	`MISRA C:2012 Rule 21.13`
MISRA C:2012 Rule 21.17	Use of the string handling function from `<string.h>` shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters	`MISRA C:2012 Rule 21.17`
MISRA C:2012 Rule 21.18	The `size_t` argument passed to any function in `<string.h>` shall have an appropriate value	`MISRA C:2012 Rule 21.18`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 21.19	The pointers returned by the Standard Library functions `localeconv`, `getenv`, `setlocale` or `strerror` shall only be used as if they have pointer to `const`-qualified type	`MISRA C:2012 Rule 21.19`
MISRA C:2012 Rule 21.20	The pointer returned by the Standard Library functions `asctime`, `ctime`, `gmtime`, `localtime`, `localeconv`, `getenv`, `setlocale` or `strerror` shall not be used following a subsequent call to the same function	`MISRA C:2012 Rule 21.20`
MISRA C:2012 Rule 22.2	A block of memory shall only be freed if it was allocated by means of a Standard Library function	`MISRA C:2012 Rule 22.2`
MISRA C:2012 Rule 22.4	There shall be no attempt to write to a stream which has been opened as read-only	`MISRA C:2012 Rule 22.4`
MISRA C:2012 Rule 22.5	A pointer to a `FILE` object shall not be dereferenced	`MISRA C:2012 Rule 22.5`
MISRA C:2012 Rule 22.6	The value of a pointer to a `FILE` shall not be used after the associated stream has been closed	`MISRA C:2012 Rule 22.6`
MISRA C:2012 Rule 9.1	The value of an object with automatic storage duration shall not be read before it has been set	`MISRA C:2012 Rule 9.1`

Required Rules

Compliant C code must follow these coding rules. The standard permits only the deviations that you formally record and authorize. Polyspace supports 110 out of 110 such rules.

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 10.1	Operands shall not be of an inappropriate essential type	`MISRA C:2012 Rule 10.1`
MISRA C:2012 Rule 10.2	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations	`MISRA C:2012 Rule 10.2`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category	`MISRA C:2012 Rule 10.3`
MISRA C:2012 Rule 10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category	`MISRA C:2012 Rule 10.4`
MISRA C:2012 Rule 10.6	The value of a composite expression shall not be assigned to an object with wider essential type	`MISRA C:2012 Rule 10.6`
MISRA C:2012 Rule 10.7	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type	`MISRA C:2012 Rule 10.7`
MISRA C:2012 Rule 10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type	`MISRA C:2012 Rule 10.8`
MISRA C:2012 Rule 11.1	Conversions shall not be performed between a pointer to a function and any other type	`MISRA C:2012 Rule 11.1`
MISRA C:2012 Rule 11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type	`MISRA C:2012 Rule 11.2`
MISRA C:2012 Rule 11.3	A cast shall not be performed between a pointer to object type and a pointer to a different object type	`MISRA C:2012 Rule 11.3`
MISRA C:2012 Rule 11.6	A cast shall not be performed between pointer to void and an arithmetic type	`MISRA C:2012 Rule 11.6`
MISRA C:2012 Rule 11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type	`MISRA C:2012 Rule 11.7`
MISRA C:2012 Rule 11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer	`MISRA C:2012 Rule 11.8`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 11.9	The macro NULL shall be the only permitted form of integer null pointer constant	`MISRA C:2012 Rule 11.9`
MISRA C:2012 Rule 12.2	The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand	`MISRA C:2012 Rule 12.2`
MISRA C:2012 Rule 13.1	Initializer lists shall not contain persistent side effects	`MISRA C:2012 Rule 13.1`
MISRA C:2012 Rule 13.2	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders	`MISRA C:2012 Rule 13.2`
MISRA C:2012 Rule 13.5	The right hand operand of a logical && or || operator shall not contain persistent side effects	`MISRA C:2012 Rule 13.5`
MISRA C:2012 Rule 14.1	A loop counter shall not have essentially floating type	`MISRA C:2012 Rule 14.1`
MISRA C:2012 Rule 14.2	A for loop shall be well-formed	`MISRA C:2012 Rule 14.2`
MISRA C:2012 Rule 14.3	Controlling expressions shall not be invariant	`MISRA C:2012 Rule 14.3`
MISRA C:2012 Rule 14.4	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type	`MISRA C:2012 Rule 14.4`
MISRA C:2012 Rule 15.2	The goto statement shall jump to a label declared later in the same function	`MISRA C:2012 Rule 15.2`
MISRA C:2012 Rule 15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement	`MISRA C:2012 Rule 15.3`
MISRA C:2012 Rule 15.6	The body of an iteration-statement or a selection-statement shall be a compound statement	`MISRA C:2012 Rule 15.6`
MISRA C:2012 Rule 15.7	All if … else if constructs shall be terminated with an else statement	`MISRA C:2012 Rule 15.7`
MISRA C:2012 Rule 16.1	All switch statements shall be well-formed	`MISRA C:2012 Rule 16.1`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 16.2	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	`MISRA C:2012 Rule 16.2`
MISRA C:2012 Rule 16.3	An unconditional break statement shall terminate every switch-clause	`MISRA C:2012 Rule 16.3`
MISRA C:2012 Rule 16.4	Every switch statement shall have a default label	`MISRA C:2012 Rule 16.4`
MISRA C:2012 Rule 16.5	A default label shall appear as either the first or the last switch label of a switch statement	`MISRA C:2012 Rule 16.5`
MISRA C:2012 Rule 16.6	Every switch statement shall have at least two switch-clauses	`MISRA C:2012 Rule 16.6`
MISRA C:2012 Rule 16.7	A switch-expression shall not have essentially Boolean type	`MISRA C:2012 Rule 16.7`
MISRA C:2012 Rule 17.1	The features of <stdarg.h> shall not be used	`MISRA C:2012 Rule 17.1`
MISRA C:2012 Rule 17.2	Functions shall not call themselves, either directly or indirectly	`MISRA C:2012 Rule 17.2`
MISRA C:2012 Rule 17.7	The value returned by a function having non-void return type shall be used	`MISRA C:2012 Rule 17.7`
MISRA C:2012 Rule 18.1	A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand	`MISRA C:2012 Rule 18.1`
MISRA C:2012 Rule 18.2	Subtraction between pointers shall only be applied to pointers that address elements of the same array	`MISRA C:2012 Rule 18.2`
MISRA C:2012 Rule 18.3	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object	`MISRA C:2012 Rule 18.3`
MISRA C:2012 Rule 18.6	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist	`MISRA C:2012 Rule 18.6`
MISRA C:2012 Rule 18.7	Flexible array members shall not be declared	`MISRA C:2012 Rule 18.7`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 18.8	Variable-length array types shall not be used	`MISRA C:2012 Rule 18.8`
MISRA C:2012 Rule 1.1	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits	`MISRA C:2012 Rule 1.1`
MISRA C:2012 Rule 1.3	There shall be no occurrence of undefined or critical unspecified behaviour	`MISRA C:2012 Rule 1.3`
MISRA C:2012 Rule 1.4	Emergent language features shall not be used	`MISRA C:2012 Rule 1.4`
MISRA C:2012 Rule 20.11	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator	`MISRA C:2012 Rule 20.11`
MISRA C:2012 Rule 20.12	A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators	`MISRA C:2012 Rule 20.12`
MISRA C:2012 Rule 20.13	A line whose first token is # shall be a valid preprocessing directive	`MISRA C:2012 Rule 20.13`
MISRA C:2012 Rule 20.14	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related	`MISRA C:2012 Rule 20.14`
MISRA C:2012 Rule 20.2	The ', " or \ characters and the /* or // character sequences shall not occur in a header file name	`MISRA C:2012 Rule 20.2`
MISRA C:2012 Rule 20.3	The #include directive shall be followed by either a <filename> or "filename" sequence	`MISRA C:2012 Rule 20.3`
MISRA C:2012 Rule 20.4	A macro shall not be defined with the same name as a keyword	`MISRA C:2012 Rule 20.4`
MISRA C:2012 Rule 20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument	`MISRA C:2012 Rule 20.6`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses	`MISRA C:2012 Rule 20.7`
MISRA C:2012 Rule 20.8	The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1	`MISRA C:2012 Rule 20.8`
MISRA C:2012 Rule 20.9	All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation	`MISRA C:2012 Rule 20.9`
MISRA C:2012 Rule 21.1	#define and #undef shall not be used on a reserved identifier or reserved macro name	`MISRA C:2012 Rule 21.1`
MISRA C:2012 Rule 21.10	The Standard Library time and date functions shall not be used	`MISRA C:2012 Rule 21.10`
MISRA C:2012 Rule 21.11	The standard header file <tgmath.h> shall not be used	`MISRA C:2012 Rule 21.11`
MISRA C:2012 Rule 21.14	The Standard Library function `memcmp` shall not be used to compare null terminated strings	`MISRA C:2012 Rule 21.14`
MISRA C:2012 Rule 21.15	The pointer arguments to the Standard Library functions `memcpy`, `memmove` and `memcmp` shall be pointers to qualified or unqualified versions of compatible types	`MISRA C:2012 Rule 21.15`
MISRA C:2012 Rule 21.16	The pointer arguments to the Standard Library function `memcmp` shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type	`MISRA C:2012 Rule 21.16`
MISRA C:2012 Rule 21.2	A reserved identifier or reserved macro name shall not be declared	`MISRA C:2012 Rule 21.2`
MISRA C:2012 Rule 21.21	The Standard Library function `system` of `<stdlib.h>` shall not be used	`MISRA C:2012 Rule 21.21`
MISRA C:2012 Rule 21.3	The memory allocation and deallocation functions of `<stdlib.h>` shall not be used	`MISRA C:2012 Rule 21.3`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 21.4	The standard header file <setjmp.h> shall not be used	`MISRA C:2012 Rule 21.4`
MISRA C:2012 Rule 21.5	The standard header file <signal.h> shall not be used	`MISRA C:2012 Rule 21.5`
MISRA C:2012 Rule 21.6	The Standard Library input/ output functions shall not be used	`MISRA C:2012 Rule 21.6`
MISRA C:2012 Rule 21.7	The Standard Library functions `atof`, `atoi`, `atol`, and `atoll` functions of `<stdlib.h>` shall not be used	`MISRA C:2012 Rule 21.7`
MISRA C:2012 Rule 21.8	The Standard Library termination functions of `<stdlib.h>` shall not be used	`MISRA C:2012 Rule 21.8`
MISRA C:2012 Rule 21.9	The Standard Library library functions `bsearch` and `qsort` of `<stdlib.h>` shall not be used	`MISRA C:2012 Rule 21.9`
MISRA C:2012 Rule 22.1	All resources obtained dynamically by means of Standard Library functions shall be explicitly released	`MISRA C:2012 Rule 22.1`
MISRA C:2012 Rule 22.10	The value of `errno` shall only be tested when the last function to be called was an `errno`-setting function	`MISRA C:2012 Rule 22.10`
MISRA C:2012 Rule 22.3	The same file shall not be open for read and write access at the same time on different streams	`MISRA C:2012 Rule 22.3`
MISRA C:2012 Rule 22.7	The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF	`MISRA C:2012 Rule 22.7`
MISRA C:2012 Rule 22.8	The value of `errno` shall be set to zero prior to a call to an `errno`-setting-function	`MISRA C:2012 Rule 22.8`
MISRA C:2012 Rule 22.9	The value of `errno` shall be tested against zero after calling an `errno`-setting function	`MISRA C:2012 Rule 22.9`
MISRA C:2012 Rule 2.1	A project shall not contain unreachable code	`MISRA C:2012 Rule 2.1`
MISRA C:2012 Rule 2.2	There shall be no dead code	`MISRA C:2012 Rule 2.2`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 3.1	The character sequences `/*` and `//` shall not be used within a comment	`MISRA C:2012 Rule 3.1`
MISRA C:2012 Rule 3.2	Line-splicing shall not be used in `//` comments	`MISRA C:2012 Rule 3.2`
MISRA C:2012 Rule 4.1	Octal and hexadecimal escape sequences shall be terminated	`MISRA C:2012 Rule 4.1`
MISRA C:2012 Rule 5.1	External identifiers shall be distinct	`MISRA C:2012 Rule 5.1`
MISRA C:2012 Rule 5.2	Identifiers declared in the same scope and name space shall be distinct	`MISRA C:2012 Rule 5.2`
MISRA C:2012 Rule 5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope	`MISRA C:2012 Rule 5.3`
MISRA C:2012 Rule 5.4	Macro identifiers shall be distinct	`MISRA C:2012 Rule 5.4`
MISRA C:2012 Rule 5.5	Identifiers shall be distinct from macro names	`MISRA C:2012 Rule 5.5`
MISRA C:2012 Rule 5.6	A typedef name shall be a unique identifier	`MISRA C:2012 Rule 5.6`
MISRA C:2012 Rule 5.7	A tag name shall be a unique identifier	`MISRA C:2012 Rule 5.7`
MISRA C:2012 Rule 5.8	Identifiers that define objects or functions with external linkage shall be unique	`MISRA C:2012 Rule 5.8`
MISRA C:2012 Rule 6.1	Bit-fields shall only be declared with an appropriate type	`MISRA C:2012 Rule 6.1`
MISRA C:2012 Rule 6.2	Single-bit named bit fields shall not be of a signed type	`MISRA C:2012 Rule 6.2`
MISRA C:2012 Rule 7.1	Octal constants shall not be used	`MISRA C:2012 Rule 7.1`
MISRA C:2012 Rule 7.2	A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type	`MISRA C:2012 Rule 7.2`
MISRA C:2012 Rule 7.3	The lowercase character “l” shall not be used in a literal suffix	`MISRA C:2012 Rule 7.3`
MISRA C:2012 Rule 7.4	A string literal shall not be assigned to an object unless the object’s type is “pointer to const-qualified char”	`MISRA C:2012 Rule 7.4`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 8.1	Types shall be explicitly specified	MISRA C:2012 Rule 8.1
MISRA C:2012 Rule 8.10	An inline function shall be declared with the static storage class	MISRA C:2012 Rule 8.10
MISRA C:2012 Rule 8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique	MISRA C:2012 Rule 8.12
MISRA C:2012 Rule 8.14	The restrict type qualifier shall not be used	MISRA C:2012 Rule 8.14
MISRA C:2012 Rule 8.2	Function types shall be in prototype form with named parameters	MISRA C:2012 Rule 8.2
MISRA C:2012 Rule 8.3	All declarations of an object or function shall use the same names and type qualifiers	MISRA C:2012 Rule 8.3
MISRA C:2012 Rule 8.4	A compatible declaration shall be visible when an object or function with external linkage is defined	MISRA C:2012 Rule 8.4
MISRA C:2012 Rule 8.5	An external object or function shall be declared once in one and only one file	MISRA C:2012 Rule 8.5
MISRA C:2012 Rule 8.6	An identifier with external linkage shall have exactly one external definition	MISRA C:2012 Rule 8.6
MISRA C:2012 Rule 8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage	MISRA C:2012 Rule 8.8
MISRA C:2012 Rule 9.2	The initializer for an aggregate or union shall be enclosed in braces	MISRA C:2012 Rule 9.2
MISRA C:2012 Rule 9.3	Arrays shall not be partially initialized	MISRA C:2012 Rule 9.3
MISRA C:2012 Rule 9.4	An element of an object shall not be initialized more than once	MISRA C:2012 Rule 9.4
MISRA C:2012 Rule 9.5	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly	MISRA C:2012 Rule 9.5

See Also

`Check MISRA C:2012 (-misra3)`

More About

- "Check for and Review Coding Standard Violations"
- "Coding Standards"
- "Decidable MISRA C:2012 Rules Supported by Polyspace Bug Finder"
- "Undecidable MISRA C:2012 Rules and Directives Supported by Polyspace Bug Finder"
- "Required and Statically Enforceable CERT C Rules Supported by Polyspace Bug Finder"
- "Checkers Deactivated in Polyspace as You Code Analysis"

Decidable MISRA C:2012 Rules Supported by Polyspace Bug Finder

The MISRA C:2012 standard classifies rules that can be statically enforced in all possible cases as **Decidable**. Polyspace supports 122 out of 122 such rules. None of the MISRA C:2012 directives are statically enforceable.

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 10.1	Operands shall not be of an inappropriate essential type	`MISRA C:2012 Rule 10.1`
MISRA C:2012 Rule 10.2	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations	`MISRA C:2012 Rule 10.2`
MISRA C:2012 Rule 10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category	`MISRA C:2012 Rule 10.3`
MISRA C:2012 Rule 10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category	`MISRA C:2012 Rule 10.4`
MISRA C:2012 Rule 10.5	The value of an expression should not be cast to an inappropriate essential type	`MISRA C:2012 Rule 10.5`
MISRA C:2012 Rule 10.6	The value of a composite expression shall not be assigned to an object with wider essential type	`MISRA C:2012 Rule 10.6`
MISRA C:2012 Rule 10.7	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type	`MISRA C:2012 Rule 10.7`
MISRA C:2012 Rule 10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type	`MISRA C:2012 Rule 10.8`
MISRA C:2012 Rule 11.1	Conversions shall not be performed between a pointer to a function and any other type	`MISRA C:2012 Rule 11.1`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type	`MISRA C:2012 Rule 11.2`
MISRA C:2012 Rule 11.3	A cast shall not be performed between a pointer to object type and a pointer to a different object type	`MISRA C:2012 Rule 11.3`
MISRA C:2012 Rule 11.4	A conversion should not be performed between a pointer to object and an integer type	`MISRA C:2012 Rule 11.4`
MISRA C:2012 Rule 11.5	A conversion should not be performed from pointer to void into pointer to object	`MISRA C:2012 Rule 11.5`
MISRA C:2012 Rule 11.6	A cast shall not be performed between pointer to void and an arithmetic type	`MISRA C:2012 Rule 11.6`
MISRA C:2012 Rule 11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type	`MISRA C:2012 Rule 11.7`
MISRA C:2012 Rule 11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer	`MISRA C:2012 Rule 11.8`
MISRA C:2012 Rule 11.9	The macro NULL shall be the only permitted form of integer null pointer constant	`MISRA C:2012 Rule 11.9`
MISRA C:2012 Rule 12.1	The precedence of operators within expressions should be made explicit	`MISRA C:2012 Rule 12.1`
MISRA C:2012 Rule 12.3	The comma operator should not be used	`MISRA C:2012 Rule 12.3`
MISRA C:2012 Rule 12.4	Evaluation of constant expressions should not lead to unsigned integer wrap-around	`MISRA C:2012 Rule 12.4`
MISRA C:2012 Rule 12.5	The `sizeof` operator shall not have an operand which is a function parameter declared as “array of type”	`MISRA C:2012 Rule 12.5`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 13.3	A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator	`MISRA C:2012 Rule 13.3`
MISRA C:2012 Rule 13.4	The result of an assignment operator should not be used	`MISRA C:2012 Rule 13.4`
MISRA C:2012 Rule 13.6	The operand of the sizeof operator shall not contain any expression which has potential side effects	`MISRA C:2012 Rule 13.6`
MISRA C:2012 Rule 14.4	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type	`MISRA C:2012 Rule 14.4`
MISRA C:2012 Rule 15.1	The goto statement should not be used	`MISRA C:2012 Rule 15.1`
MISRA C:2012 Rule 15.2	The goto statement shall jump to a label declared later in the same function	`MISRA C:2012 Rule 15.2`
MISRA C:2012 Rule 15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement	`MISRA C:2012 Rule 15.3`
MISRA C:2012 Rule 15.4	There should be no more than one break or goto statement used to terminate any iteration statement	`MISRA C:2012 Rule 15.4`
MISRA C:2012 Rule 15.5	A function should have a single point of exit at the end	`MISRA C:2012 Rule 15.5`
MISRA C:2012 Rule 15.6	The body of an iteration-statement or a selection-statement shall be a compound statement	`MISRA C:2012 Rule 15.6`
MISRA C:2012 Rule 15.7	All if ... else if constructs shall be terminated with an else statement	`MISRA C:2012 Rule 15.7`
MISRA C:2012 Rule 16.1	All switch statements shall be well-formed	`MISRA C:2012 Rule 16.1`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 16.2	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	`MISRA C:2012 Rule 16.2`
MISRA C:2012 Rule 16.3	An unconditional break statement shall terminate every switch-clause	`MISRA C:2012 Rule 16.3`
MISRA C:2012 Rule 16.4	Every switch statement shall have a default label	`MISRA C:2012 Rule 16.4`
MISRA C:2012 Rule 16.5	A default label shall appear as either the first or the last switch label of a switch statement	`MISRA C:2012 Rule 16.5`
MISRA C:2012 Rule 16.6	Every switch statement shall have at least two switch-clauses	`MISRA C:2012 Rule 16.6`
MISRA C:2012 Rule 16.7	A switch-expression shall not have essentially Boolean type	`MISRA C:2012 Rule 16.7`
MISRA C:2012 Rule 17.1	The features of <stdarg.h> shall not be used	`MISRA C:2012 Rule 17.1`
MISRA C:2012 Rule 17.3	A function shall not be declared implicitly	`MISRA C:2012 Rule 17.3`
MISRA C:2012 Rule 17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression	`MISRA C:2012 Rule 17.4`
MISRA C:2012 Rule 17.6	The declaration of an array parameter shall not contain the static keyword between the [ ]	`MISRA C:2012 Rule 17.6`
MISRA C:2012 Rule 17.7	The value returned by a function having non-void return type shall be used	`MISRA C:2012 Rule 17.7`
MISRA C:2012 Rule 18.4	The +, -, += and -= operators should not be applied to an expression of pointer type	`MISRA C:2012 Rule 18.4`
MISRA C:2012 Rule 18.5	Declarations should contain no more than two levels of pointer nesting	`MISRA C:2012 Rule 18.5`
MISRA C:2012 Rule 18.7	Flexible array members shall not be declared	`MISRA C:2012 Rule 18.7`
MISRA C:2012 Rule 18.8	Variable-length array types shall not be used	`MISRA C:2012 Rule 18.8`
MISRA C:2012 Rule 19.2	The union keyword should not be used	`MISRA C:2012 Rule 19.2`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 1.1	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation’s translation limits	`MISRA C:2012 Rule 1.1`
MISRA C:2012 Rule 1.2	Language extensions should not be used	`MISRA C:2012 Rule 1.2`
MISRA C:2012 Rule 1.4	Emergent language features shall not be used	`MISRA C:2012 Rule 1.4`
MISRA C:2012 Rule 20.1	#include directives should only be preceded by preprocessor directives or comments	`MISRA C:2012 Rule 20.1`
MISRA C:2012 Rule 20.10	The # and ## preprocessor operators should not be used	`MISRA C:2012 Rule 20.10`
MISRA C:2012 Rule 20.11	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator	`MISRA C:2012 Rule 20.11`
MISRA C:2012 Rule 20.12	A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators	`MISRA C:2012 Rule 20.12`
MISRA C:2012 Rule 20.13	A line whose first token is # shall be a valid preprocessing directive	`MISRA C:2012 Rule 20.13`
MISRA C:2012 Rule 20.14	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related	`MISRA C:2012 Rule 20.14`
MISRA C:2012 Rule 20.2	The ', " or \ characters and the /* or // character sequences shall not occur in a header file name	`MISRA C:2012 Rule 20.2`
MISRA C:2012 Rule 20.3	The #include directive shall be followed by either a <filename> or "filename" sequence	`MISRA C:2012 Rule 20.3`
MISRA C:2012 Rule 20.4	A macro shall not be defined with the same name as a keyword	`MISRA C:2012 Rule 20.4`
MISRA C:2012 Rule 20.5	#undef should not be used	`MISRA C:2012 Rule 20.5`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument	`MISRA C:2012 Rule 20.6`
MISRA C:2012 Rule 20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses	`MISRA C:2012 Rule 20.7`
MISRA C:2012 Rule 20.8	The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1	`MISRA C:2012 Rule 20.8`
MISRA C:2012 Rule 20.9	All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation	`MISRA C:2012 Rule 20.9`
MISRA C:2012 Rule 21.1	#define and #undef shall not be used on a reserved identifier or reserved macro name	`MISRA C:2012 Rule 21.1`
MISRA C:2012 Rule 21.10	The Standard Library time and date functions shall not be used	`MISRA C:2012 Rule 21.10`
MISRA C:2012 Rule 21.11	The standard header file <tgmath.h> shall not be used	`MISRA C:2012 Rule 21.11`
MISRA C:2012 Rule 21.12	The exception handling features of `<fenv.h>` should not be used	`MISRA C:2012 Rule 21.12`
MISRA C:2012 Rule 21.15	The pointer arguments to the Standard Library functions `memcpy`, `memmove` and `memcmp` shall be pointers to qualified or unqualified versions of compatible types	`MISRA C:2012 Rule 21.15`
MISRA C:2012 Rule 21.16	The pointer arguments to the Standard Library function `memcmp` shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type	`MISRA C:2012 Rule 21.16`
MISRA C:2012 Rule 21.2	A reserved identifier or reserved macro name shall not be declared	`MISRA C:2012 Rule 21.2`
MISRA C:2012 Rule 21.21	The Standard Library function `system` of `<stdlib.h>` shall not be used	`MISRA C:2012 Rule 21.21`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 21.3	The memory allocation and deallocation functions of <stdlib.h> shall not be used	MISRA C:2012 Rule 21.3
MISRA C:2012 Rule 21.4	The standard header file <setjmp.h> shall not be used	MISRA C:2012 Rule 21.4
MISRA C:2012 Rule 21.5	The standard header file <signal.h> shall not be used	MISRA C:2012 Rule 21.5
MISRA C:2012 Rule 21.6	The Standard Library input/ output functions shall not be used	MISRA C:2012 Rule 21.6
MISRA C:2012 Rule 21.7	The Standard Library functions atof, atoi, atol, and atoll functions of <stdlib.h> shall not be used	MISRA C:2012 Rule 21.7
MISRA C:2012 Rule 21.8	The Standard Library termination functions of <stdlib.h> shall not be used	MISRA C:2012 Rule 21.8
MISRA C:2012 Rule 21.9	The Standard Library library functions bsearch and qsort of <stdlib.h> shall not be used	MISRA C:2012 Rule 21.9
MISRA C:2012 Rule 2.3	A project should not contain unused type declarations	MISRA C:2012 Rule 2.3
MISRA C:2012 Rule 2.4	A project should not contain unused tag declarations	MISRA C:2012 Rule 2.4
MISRA C:2012 Rule 2.5	A project should not contain unused macro declarations	MISRA C:2012 Rule 2.5
MISRA C:2012 Rule 2.6	A function should not contain unused label declarations	MISRA C:2012 Rule 2.6
MISRA C:2012 Rule 2.7	There should be no unused parameters in functions	MISRA C:2012 Rule 2.7
MISRA C:2012 Rule 3.1	The character sequences /* and // shall not be used within a comment	MISRA C:2012 Rule 3.1
MISRA C:2012 Rule 3.2	Line-splicing shall not be used in // comments	MISRA C:2012 Rule 3.2
MISRA C:2012 Rule 4.1	Octal and hexadecimal escape sequences shall be terminated	MISRA C:2012 Rule 4.1
MISRA C:2012 Rule 4.2	Trigraphs should not be used	MISRA C:2012 Rule 4.2
MISRA C:2012 Rule 5.1	External identifiers shall be distinct	MISRA C:2012 Rule 5.1
MISRA C:2012 Rule 5.2	Identifiers declared in the same scope and name space shall be distinct	MISRA C:2012 Rule 5.2

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope	`MISRA C:2012 Rule 5.3`
MISRA C:2012 Rule 5.4	Macro identifiers shall be distinct	`MISRA C:2012 Rule 5.4`
MISRA C:2012 Rule 5.5	Identifiers shall be distinct from macro names	`MISRA C:2012 Rule 5.5`
MISRA C:2012 Rule 5.6	A typedef name shall be a unique identifier	`MISRA C:2012 Rule 5.6`
MISRA C:2012 Rule 5.7	A tag name shall be a unique identifier	`MISRA C:2012 Rule 5.7`
MISRA C:2012 Rule 5.8	Identifiers that define objects or functions with external linkage shall be unique	`MISRA C:2012 Rule 5.8`
MISRA C:2012 Rule 5.9	Identifiers that define objects or functions with internal linkage should be unique	`MISRA C:2012 Rule 5.9`
MISRA C:2012 Rule 6.1	Bit-fields shall only be declared with an appropriate type	`MISRA C:2012 Rule 6.1`
MISRA C:2012 Rule 6.2	Single-bit named bit fields shall not be of a signed type	`MISRA C:2012 Rule 6.2`
MISRA C:2012 Rule 7.1	Octal constants shall not be used	`MISRA C:2012 Rule 7.1`
MISRA C:2012 Rule 7.2	A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type	`MISRA C:2012 Rule 7.2`
MISRA C:2012 Rule 7.3	The lowercase character “l” shall not be used in a literal suffix	`MISRA C:2012 Rule 7.3`
MISRA C:2012 Rule 7.4	A string literal shall not be assigned to an object unless the object’s type is “pointer to const-qualified char”	`MISRA C:2012 Rule 7.4`
MISRA C:2012 Rule 8.1	Types shall be explicitly specified	`MISRA C:2012 Rule 8.1`
MISRA C:2012 Rule 8.10	An inline function shall be declared with the static storage class	`MISRA C:2012 Rule 8.10`
MISRA C:2012 Rule 8.11	When an array with external linkage is declared, its size should be explicitly specified	`MISRA C:2012 Rule 8.11`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique	`MISRA C:2012 Rule 8.12`
MISRA C:2012 Rule 8.14	The restrict type qualifier shall not be used	`MISRA C:2012 Rule 8.14`
MISRA C:2012 Rule 8.2	Function types shall be in prototype form with named parameters	`MISRA C:2012 Rule 8.2`
MISRA C:2012 Rule 8.3	All declarations of an object or function shall use the same names and type qualifiers	`MISRA C:2012 Rule 8.3`
MISRA C:2012 Rule 8.4	A compatible declaration shall be visible when an object or function with external linkage is defined	`MISRA C:2012 Rule 8.4`
MISRA C:2012 Rule 8.5	An external object or function shall be declared once in one and only one file	`MISRA C:2012 Rule 8.5`
MISRA C:2012 Rule 8.6	An identifier with external linkage shall have exactly one external definition	`MISRA C:2012 Rule 8.6`
MISRA C:2012 Rule 8.7	Functions and objects should not be defined with external linkage if they are referenced in only one translation unit	`MISRA C:2012 Rule 8.7`
MISRA C:2012 Rule 8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage	`MISRA C:2012 Rule 8.8`
MISRA C:2012 Rule 8.9	An object should be defined at block scope if its identifier only appears in a single function	`MISRA C:2012 Rule 8.9`
MISRA C:2012 Rule 9.2	The initializer for an aggregate or union shall be enclosed in braces	`MISRA C:2012 Rule 9.2`
MISRA C:2012 Rule 9.3	Arrays shall not be partially initialized	`MISRA C:2012 Rule 9.3`
MISRA C:2012 Rule 9.4	An element of an object shall not be initialized more than once	`MISRA C:2012 Rule 9.4`
MISRA C:2012 Rule 9.5	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly	`MISRA C:2012 Rule 9.5`

See Also

`Check MISRA C:2012 (-misra3)`

More About

- “Check for and Review Coding Standard Violations”
- “Coding Standards”
- “Required or Mandatory MISRA C:2012 Rules Supported by Polyspace Bug Finder”
- “Required and Statically Enforceable CERT C Rules Supported by Polyspace Bug Finder”
- “Checkers Deactivated in Polyspace as You Code Analysis”
- “Undecidable MISRA C:2012 Rules and Directives Supported by Polyspace Bug Finder”

Undecidable MISRA C:2012 Rules and Directives Supported by Polyspace Bug Finder

The MISRA C:2012 standard classifies rules and directives that cannot be statically enforced in every possible cases as **Undecidable**. Polyspace supports 36 out of 36 such rules, and 15 out of 17 such directives.

Undecidable Rules

A rule is undecidable if a static analysis tool can check compliance to it only in certain cases. Polyspace shows the subset of all possible issues. For details about which issues Polyspace detects for a particular rule, see the **Polyspace Implementation** section in the reference page of the rule. Polyspace supports 36 out of 36 such rules.

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 12.2	The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand	`MISRA C:2012 Rule 12.2`
MISRA C:2012 Rule 13.1	Initializer lists shall not contain persistent side effects	`MISRA C:2012 Rule 13.1`
MISRA C:2012 Rule 13.2	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders	`MISRA C:2012 Rule 13.2`
MISRA C:2012 Rule 13.5	The right hand operand of a logical && or || operator shall not contain persistent side effects	`MISRA C:2012 Rule 13.5`
MISRA C:2012 Rule 14.1	A loop counter shall not have essentially floating type	`MISRA C:2012 Rule 14.1`
MISRA C:2012 Rule 14.2	A for loop shall be well-formed	`MISRA C:2012 Rule 14.2`
MISRA C:2012 Rule 14.3	Controlling expressions shall not be invariant	`MISRA C:2012 Rule 14.3`
MISRA C:2012 Rule 17.2	Functions shall not call themselves, either directly or indirectly	`MISRA C:2012 Rule 17.2`
MISRA C:2012 Rule 17.5	The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements	`MISRA C:2012 Rule 17.5`
MISRA C:2012 Rule 17.8	A function parameter should not be modified	`MISRA C:2012 Rule 17.8`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 18.1	A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand	`MISRA C:2012 Rule 18.1`
MISRA C:2012 Rule 18.2	Subtraction between pointers shall only be applied to pointers that address elements of the same array	`MISRA C:2012 Rule 18.2`
MISRA C:2012 Rule 18.3	The relational operators `>`, `>=`, `<` and `<=` shall not be applied to objects of pointer type except where they point into the same object	`MISRA C:2012 Rule 18.3`
MISRA C:2012 Rule 18.6	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist	`MISRA C:2012 Rule 18.6`
MISRA C:2012 Rule 19.1	An object shall not be assigned or copied to an overlapping object	`MISRA C:2012 Rule 19.1`
MISRA C:2012 Rule 1.3	There shall be no occurrence of undefined or critical unspecified behaviour	`MISRA C:2012 Rule 1.3`
MISRA C:2012 Rule 21.13	Any value passed to a function in `<ctype.h>` shall be representable as an `unsigned char` or be the value EOF	`MISRA C:2012 Rule 21.13`
MISRA C:2012 Rule 21.14	The Standard Library function `memcmp` shall not be used to compare null terminated strings	`MISRA C:2012 Rule 21.14`
MISRA C:2012 Rule 21.17	Use of the string handling function from `<string.h>` shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters	`MISRA C:2012 Rule 21.17`
MISRA C:2012 Rule 21.18	The `size_t` argument passed to any function in `<string.h>` shall have an appropriate value	`MISRA C:2012 Rule 21.18`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 21.19	The pointers returned by the Standard Library functions `localeconv`, `getenv`, `setlocale` or `strerror` shall only be used as if they have pointer to `const`-qualified type	`MISRA C:2012 Rule 21.19`
MISRA C:2012 Rule 21.20	The pointer returned by the Standard Library functions `asctime`, `ctime`, `gmtime`, `localtime`, `localeconv`, `getenv`, `setlocale` or `strerror` shall not be used following a subsequent call to the same function	`MISRA C:2012 Rule 21.20`
MISRA C:2012 Rule 22.1	All resources obtained dynamically by means of Standard Library functions shall be explicitly released	`MISRA C:2012 Rule 22.1`
MISRA C:2012 Rule 22.10	The value of `errno` shall only be tested when the last function to be called was an `errno`-setting function	`MISRA C:2012 Rule 22.10`
MISRA C:2012 Rule 22.2	A block of memory shall only be freed if it was allocated by means of a Standard Library function	`MISRA C:2012 Rule 22.2`
MISRA C:2012 Rule 22.3	The same file shall not be open for read and write access at the same time on different streams	`MISRA C:2012 Rule 22.3`
MISRA C:2012 Rule 22.4	There shall be no attempt to write to a stream which has been opened as read-only	`MISRA C:2012 Rule 22.4`
MISRA C:2012 Rule 22.5	A pointer to a `FILE` object shall not be dereferenced	`MISRA C:2012 Rule 22.5`
MISRA C:2012 Rule 22.6	The value of a pointer to a `FILE` shall not be used after the associated stream has been closed	`MISRA C:2012 Rule 22.6`
MISRA C:2012 Rule 22.7	The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF	`MISRA C:2012 Rule 22.7`
MISRA C:2012 Rule 22.8	The value of `errno` shall be set to zero prior to a call to an `errno`-setting-function	`MISRA C:2012 Rule 22.8`

MISRA C:2012 Rule	Description	Polyspace Checker
MISRA C:2012 Rule 22.9	The value of `errno` shall be tested against zero after calling an `errno`-setting function	`MISRA C:2012 Rule 22.9`
MISRA C:2012 Rule 2.1	A project shall not contain unreachable code	`MISRA C:2012 Rule 2.1`
MISRA C:2012 Rule 2.2	There shall be no dead code	`MISRA C:2012 Rule 2.2`
MISRA C:2012 Rule 8.13	A pointer should point to a const-qualified type whenever possible	`MISRA C:2012 Rule 8.13`
MISRA C:2012 Rule 9.1	The value of an object with automatic storage duration shall not be read before it has been set	`MISRA C:2012 Rule 9.1`

Undecidable Directives

A directive is undecidable if a static analysis tool can check compliance to it only in certain cases. Polyspace shows the subset of all possible issues. For details about which issues Polyspace detects for a particular directive, see the **Polyspace Implementation** section in the reference page of the directive. Polyspace supports 15 out of 17 such directives.

MISRA C:2012 Directives	Description	Polyspace Checker
MISRA C:2012 Dir 1.1	Any implementation-defined behavior on which the output of the program depends shall be documented and understood	`MISRA C:2012 Dir 1.1`
MISRA C:2012 Dir 2.1	All source files shall compile without any compilation errors	`MISRA C:2012 Dir 2.1`
MISRA C:2012 Dir 4.1	Run-time failures shall be minimized	`MISRA C:2012 Dir 4.1`
MISRA C:2012 Dir 4.10	Precautions shall be taken in order to prevent the contents of a header file being included more than once	`MISRA C:2012 Dir 4.10`
MISRA C:2012 Dir 4.11	The validity of values passed to library functions shall be checked	`MISRA C:2012 Dir 4.11`
MISRA C:2012 Dir 4.12	Dynamic memory allocation shall not be used	`MISRA C:2012 Dir 4.12`
MISRA C:2012 Dir 4.13	Functions which are designed to provide operations on a resource should be called in an appropriate sequence	`MISRA C:2012 Dir 4.13`

MISRA C:2012 Directives	Description	Polyspace Checker
MISRA C:2012 Dir 4.14	The validity of values received from external sources shall be checked	`MISRA C:2012 Dir 4.14`
MISRA C:2012 Dir 4.3	Assembly language shall be encapsulated and isolated	`MISRA C:2012 Dir 4.3`
MISRA C:2012 Dir 4.4	Sections of code should not be "commented out"	`MISRA C:2012 Dir 4.4`
MISRA C:2012 Dir 4.5	Identifiers in the same name space with overlapping visibility should be typographically unambiguous	`MISRA C:2012 Dir 4.5`
MISRA C:2012 Dir 4.6	`typedefs` that indicate size and signedness should be used in place of the basic numerical types	`MISRA C:2012 Dir 4.6`
MISRA C:2012 Dir 4.7	If a function returns error information, then that error information shall be tested	`MISRA C:2012 Dir 4.7`
MISRA C:2012 Dir 4.8	If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden	`MISRA C:2012 Dir 4.8`
MISRA C:2012 Dir 4.9	A function should be used in preference to a function-like macro where they are interchangeable	`MISRA C:2012 Dir 4.9`

See Also

`Check MISRA C:2012 (-misra3)`

More About

- “Check for and Review Coding Standard Violations”
- “Coding Standards”
- “Decidable MISRA C:2012 Rules Supported by Polyspace Bug Finder”
- “Required and Statically Enforceable CERT C Rules Supported by Polyspace Bug Finder”
- “Checkers Deactivated in Polyspace as You Code Analysis”

Essential Types in MISRA C:2012 Rules 10.x

Note Starting in a future release, Code Prover will not support checking compliance with external coding standards and calculating code metrics. Migrate to Bug Finder for these workflows. See “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	`bool` or `_Bool` (defined in `stdbool.h`) If you define a boolean type through a `typedef`, you must specify this type name before coding rules checking. For more information, see `Effective boolean types (-boolean-types)`..
Essentially character	`char`
Essentially enum	named `enum`
Essentially signed	signed `char`, signed `short`, signed `int`, signed `long`, signed `long long`
Essentially unsigned	unsigned `char`, unsigned `short`, unsigned `int`, unsigned `long`, unsigned `long long`
Essentially floating	`float`, `double`, `long double`

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 and Review Coding Standard Violations” on page 16-2
- “MISRA C:2012 Directives and Rules”

Unsupported MISRA C:2012 Guidelines

The Polyspace coding rules checkers do not check the following MISRA C:2012 directives. These directives cannot be enforced because they are outside the scope of Polyspace software. These guidelines concern documentation, dynamic aspects, or functional aspects of MISRA rules.

For the list of supported rules and directives, see “MISRA C:2012 Directives and Rules”.

Number	Category	AGC Category	Definition
Directive 3.1	Required	Required	All code shall be traceable to documented requirements
Directive 4.2	Advisory	Advisory	All usage of assembly language should be documented

See Also

More About

- “MISRA C:2012 Directives and Rules”

Required and Statically Enforceable CERT C Rules Supported by Polyspace Bug Finder

The CERT C standard classify the guidelines that compliant C code must follow as **Rules**. These **Rules** are also considered enforceable by static analysis. Polyspace supports 120 out of 120 such guidelines.

CERT C Rule	Description	Polyspace Checker
CERT C: Rule ARR30-C	Do not form or use out-of-bounds pointers or array subscripts	`CERT C: Rule ARR30-C`
CERT C: Rule ARR32-C	Ensure size arguments for variable length arrays are in a valid range	`CERT C: Rule ARR32-C`
CERT C: Rule ARR36-C	Do not subtract or compare two pointers that do not refer to the same array	`CERT C: Rule ARR36-C`
CERT C: Rule ARR37-C	Do not add or subtract an integer to a pointer to a non-array object	`CERT C: Rule ARR37-C`
CERT C: Rule ARR38-C	Guarantee that library functions do not form invalid pointers	`CERT C: Rule ARR38-C`
CERT C: Rule ARR39-C	Do not add or subtract a scaled integer to a pointer	`CERT C: Rule ARR39-C`
CERT C: Rule CON30-C	Clean up thread-specific storage	`CERT C: Rule CON30-C`
CERT C: Rule CON31-C	Do not destroy a mutex while it is locked	`CERT C: Rule CON31-C`
CERT C: Rule CON32-C	Prevent data races when accessing bit fields from multiple threads	`CERT C: Rule CON32-C`
CERT C: Rule CON33-C	Avoid race conditions when using library functions	`CERT C: Rule CON33-C`
CERT C: Rule CON34-C	Declare objects shared between threads with appropriate storage durations	`CERT C: Rule CON34-C`
CERT C: Rule CON35-C	Avoid deadlock by locking in a predefined order	`CERT C: Rule CON35-C`
CERT C: Rule CON36-C	Wrap functions that can spuriously wake up in a loop	`CERT C: Rule CON36-C`
CERT C: Rule CON37-C	Do not call signal() in a multithreaded program	`CERT C: Rule CON37-C`
CERT C: Rule CON38-C	Preserve thread safety and liveness when using condition variables	`CERT C: Rule CON38-C`

CERT C Rule	Description	Polyspace Checker
CERT C: Rule CON39-C	Do not join or detach a thread that was previously joined or detached	`CERT C: Rule CON39-C`
CERT C: Rule CON40-C	Do not refer to an atomic variable twice in an expression	`CERT C: Rule CON40-C`
CERT C: Rule CON41-C	Wrap functions that can fail spuriously in a loop	`CERT C: Rule CON41-C`
CERT C: Rule CON43-C	Do not allow data races in multithreaded code	`CERT C: Rule CON43-C`
CERT C: Rule DCL30-C	Declare objects with appropriate storage durations	`CERT C: Rule DCL30-C`
CERT C: Rule DCL31-C	Declare identifiers before using them	`CERT C: Rule DCL31-C`
CERT C: Rule DCL36-C	Do not declare an identifier with conflicting linkage classifications	`CERT C: Rule DCL36-C`
CERT C: Rule DCL37-C	Do not declare or define a reserved identifier	`CERT C: Rule DCL37-C`
CERT C: Rule DCL38-C	Use the correct syntax when declaring a flexible array member	`CERT C: Rule DCL38-C`
CERT C: Rule DCL39-C	Avoid information leakage in structure padding	`CERT C: Rule DCL39-C`
CERT C: Rule DCL40-C	Do not create incompatible declarations of the same function or object	`CERT C: Rule DCL40-C`
CERT C: Rule DCL41-C	Do not declare variables inside a switch statement before the first case label	`CERT C: Rule DCL41-C`
CERT C: Rule ENV30-C	Do not modify the object referenced by the return value of certain functions	`CERT C: Rule ENV30-C`
CERT C: Rule ENV31-C	Do not rely on an environment pointer following an operation that may invalidate it	`CERT C: Rule ENV31-C`
CERT C: Rule ENV32-C	All exit handlers must return normally	`CERT C: Rule ENV32-C`
CERT C: Rule ENV33-C	Do not call system()	`CERT C: Rule ENV33-C`
CERT C: Rule ENV34-C	Do not store pointers returned by certain functions	`CERT C: Rule ENV34-C`

CERT C Rule	Description	Polyspace Checker
CERT C: Rule ERR30-C	Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure	CERT C: Rule ERR30-C
CERT C: Rule ERR32-C	Do not rely on indeterminate values of errno	CERT C: Rule ERR32-C
CERT C: Rule ERR33-C	Detect and handle standard library errors	CERT C: Rule ERR33-C
CERT C: Rule ERR34-C	Detect errors when converting a string to a number	CERT C: Rule ERR34-C
CERT C: Rule EXP30-C	Do not depend on the order of evaluation for side effects	CERT C: Rule EXP30-C
CERT C: Rule EXP32-C	Do not access a volatile object through a nonvolatile reference	CERT C: Rule EXP32-C
CERT C: Rule EXP33-C	Do not read uninitialized memory	CERT C: Rule EXP33-C
CERT C: Rule EXP34-C	Do not dereference null pointers	CERT C: Rule EXP34-C
CERT C: Rule EXP35-C	Do not modify objects with temporary lifetime	CERT C: Rule EXP35-C
CERT C: Rule EXP36-C	Do not cast pointers into more strictly aligned pointer types	CERT C: Rule EXP36-C
CERT C: Rule EXP37-C	Call functions with the correct number and type of arguments	CERT C: Rule EXP37-C
CERT C: Rule EXP39-C	Do not access a variable through a pointer of an incompatible type	CERT C: Rule EXP39-C
CERT C: Rule EXP40-C	Do not modify constant objects	CERT C: Rule EXP40-C
CERT C: Rule EXP42-C	Do not compare padding data	CERT C: Rule EXP42-C
CERT C: Rule EXP43-C	Avoid undefined behavior when using restrict-qualified pointers	CERT C: Rule EXP43-C
CERT C: Rule EXP44-C	Do not rely on side effects in operands to sizeof, _Alignof, or _Generic	CERT C: Rule EXP44-C
CERT C: Rule EXP45-C	Do not perform assignments in selection statements	CERT C: Rule EXP45-C
CERT C: Rule EXP46-C	Do not use a bitwise operator with a Boolean-like operand	CERT C: Rule EXP46-C
CERT C: Rule EXP47-C	Do not call va_arg with an argument of the incorrect type	CERT C: Rule EXP47-C
CERT C: Rule FIO30-C	Exclude user input from format strings	CERT C: Rule FIO30-C

CERT C Rule	Description	Polyspace Checker
CERT C: Rule FIO32-C	Do not perform operations on devices that are only appropriate for files	`CERT C: Rule FIO32-C`
CERT C: Rule FIO34-C	Distinguish between characters read from a file and EOF or WEOF	`CERT C: Rule FIO34-C`
CERT C: Rule FIO37-C	Do not assume that fgets() or fgetws() returns a nonempty string when successful	`CERT C: Rule FIO37-C`
CERT C: Rule FIO38-C	Do not copy a FILE object	`CERT C: Rule FIO38-C`
CERT C: Rule FIO39-C	Do not alternately input and output from a stream without an intervening flush or positioning call	`CERT C: Rule FIO39-C`
CERT C: Rule FIO40-C	Reset strings on fgets() or fgetws() failure	`CERT C: Rule FIO40-C`
CERT C: Rule FIO41-C	Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects	`CERT C: Rule FIO41-C`
CERT C: Rule FIO42-C	Close files when they are no longer needed	`CERT C: Rule FIO42-C`
CERT C: Rule FIO44-C	Only use values for fsetpos() that are returned from fgetpos()	`CERT C: Rule FIO44-C`
CERT C: Rule FIO45-C	Avoid TOCTOU race conditions while accessing files	`CERT C: Rule FIO45-C`
CERT C: Rule FIO46-C	Do not access a closed file	`CERT C: Rule FIO46-C`
CERT C: Rule FIO47-C	Use valid format strings	`CERT C: Rule FIO47-C`
CERT C: Rule FLP30-C	Do not use floating-point variables as loop counters	`CERT C: Rule FLP30-C`
CERT C: Rule FLP32-C	Prevent or detect domain and range errors in math functions	`CERT C: Rule FLP32-C`
CERT C: Rule FLP34-C	Ensure that floating-point conversions are within range of the new type	`CERT C: Rule FLP34-C`
CERT C: Rule FLP36-C	Preserve precision when converting integral values to floating-point type	`CERT C: Rule FLP36-C`
CERT C: Rule FLP37-C	Do not use object representations to compare floating-point values	`CERT C: Rule FLP37-C`
CERT C: Rule INT30-C	Ensure that unsigned integer operations do not wrap	`CERT C: Rule INT30-C`

CERT C Rule	Description	Polyspace Checker
CERT C: Rule INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data	`CERT C: Rule INT31-C`
CERT C: Rule INT32-C	Ensure that operations on signed integers do not result in overflow	`CERT C: Rule INT32-C`
CERT C: Rule INT33-C	Ensure that division and remainder operations do not result in divide-by-zero errors	`CERT C: Rule INT33-C`
CERT C: Rule INT34-C	Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand	`CERT C: Rule INT34-C`
CERT C: Rule INT35-C	Use correct integer precisions	`CERT C: Rule INT35-C`
CERT C: Rule INT36-C	Converting a pointer to integer or integer to pointer	`CERT C: Rule INT36-C`
CERT C: Rule MEM30-C	Do not access freed memory	`CERT C: Rule MEM30-C`
CERT C: Rule MEM31-C	Free dynamically allocated memory when no longer needed	`CERT C: Rule MEM31-C`
CERT C: Rule MEM33-C	Allocate and copy structures containing a flexible array member dynamically	`CERT C: Rule MEM33-C`
CERT C: Rule MEM34-C	Only free memory allocated dynamically	`CERT C: Rule MEM34-C`
CERT C: Rule MEM35-C	Allocate sufficient memory for an object	`CERT C: Rule MEM35-C`
CERT C: Rule MEM36-C	Do not modify the alignment of objects by calling realloc()	`CERT C: Rule MEM36-C`
CERT C: Rule MSC30-C	Do not use the rand() function for generating pseudorandom numbers	`CERT C: Rule MSC30-C`
CERT C: Rule MSC32-C	Properly seed pseudorandom number generators	`CERT C: Rule MSC32-C`
CERT C: Rule MSC33-C	Do not pass invalid data to the asctime() function	`CERT C: Rule MSC33-C`
CERT C: Rule MSC37-C	Ensure that control never reaches the end of a non-void function	`CERT C: Rule MSC37-C`
CERT C: Rule MSC38-C	Do not treat a predefined identifier as an object if it might only be implemented as a macro	`CERT C: Rule MSC38-C`
CERT C: Rule MSC39-C	Do not call va_arg() on a va_list that has an indeterminate value	`CERT C: Rule MSC39-C`

CERT C Rule	Description	Polyspace Checker
CERT C: Rule MSC40-C	Do not violate constraints	`CERT C: Rule MSC40-C`
CERT C: Rule MSC41-C	Never hard code sensitive information	`CERT C: Rule MSC41-C`
CERT C: Rule POS30-C	Use the readlink() function properly	`CERT C: Rule POS30-C`
CERT C: Rule POS34-C	Do not call putenv() with a pointer to an automatic variable as the argument	`CERT C: Rule POS34-C`
CERT C: Rule POS35-C	Avoid race conditions while checking for the existence of a symbolic link	`CERT C: Rule POS35-C`
CERT C: Rule POS36-C	Observe correct revocation order while relinquishing privileges	`CERT C: Rule POS36-C`
CERT C: Rule POS37-C	Ensure that privilege relinquishment is successful	`CERT C: Rule POS37-C`
CERT C: Rule POS38-C	Beware of race conditions when using fork and file descriptors	`CERT C: Rule POS38-C`
CERT C: Rule POS39-C	Use the correct byte ordering when transferring data between systems	`CERT C: Rule POS39-C`
CERT C: Rule POS44-C	Do not use signals to terminate threads	`CERT C: Rule POS44-C`
CERT C: Rule POS47-C	Do not use threads that can be canceled asynchronously	`CERT C: Rule POS47-C`
CERT C: Rule POS48-C	Do not unlock or destroy another POSIX thread's mutex	`CERT C: Rule POS48-C`
CERT C: Rule POS49-C	When data must be accessed by multiple threads, provide a mutex and guarantee no adjacent data is also accessed	`CERT C: Rule POS49-C`
CERT C: Rule POS50-C	Declare objects shared between POSIX threads with appropriate storage durations	`CERT C: Rule POS50-C`
CERT C: Rule POS51-C	Avoid deadlock with POSIX threads by locking in predefined order	`CERT C: Rule POS51-C`
CERT C: Rule POS52-C	Do not perform operations that can block while holding a POSIX lock	`CERT C: Rule POS52-C`
CERT C: Rule POS53-C	Do not use more than one mutex for concurrent waiting operations on a condition variable	`CERT C: Rule POS53-C`

CERT C Rule	Description	Polyspace Checker
CERT C: Rule POS54-C	Detect and handle POSIX library errors	`CERT C: Rule POS54-C`
CERT C: Rule PRE30-C	Do not create a universal character name through concatenation	`CERT C: Rule PRE30-C`
CERT C: Rule PRE31-C	Avoid side effects in arguments to unsafe macros	`CERT C: Rule PRE31-C`
CERT C: Rule PRE32-C	Do not use preprocessor directives in invocations of function-like macros	`CERT C: Rule PRE32-C`
CERT C: Rule SIG30-C	Call only asynchronous-safe functions within signal handlers	`CERT C: Rule SIG30-C`
CERT C: Rule SIG31-C	Do not access shared objects in signal handlers	`CERT C: Rule SIG31-C`
CERT C: Rule SIG34-C	Do not call signal() from within interruptible signal handlers	`CERT C: Rule SIG34-C`
CERT C: Rule SIG35-C	Do not return from a computational exception signal handler	`CERT C: Rule SIG35-C`
CERT C: Rule STR30-C	Do not attempt to modify string literals	`CERT C: Rule STR30-C`
CERT C: Rule STR31-C	Guarantee that storage for strings has sufficient space for character data and the null terminator	`CERT C: Rule STR31-C`
CERT C: Rule STR32-C	Do not pass a non-null-terminated character sequence to a library function that expects a string	`CERT C: Rule STR32-C`
CERT C: Rule STR34-C	Cast characters to unsigned char before converting to larger integer sizes	`CERT C: Rule STR34-C`
CERT C: Rule STR37-C	Arguments to character-handling functions must be representable as an unsigned char	`CERT C: Rule STR37-C`
CERT C: Rule STR38-C	Do not confuse narrow and wide character strings and functions	`CERT C: Rule STR38-C`
CERT C: Rule WIN30-C	Properly pair allocation and deallocation functions	`CERT C: Rule WIN30-C`

See Also

`Check SEI CERT-C (-cert-c)`

More About

- “Check for and Review Coding Standard Violations”
- “Coding Standards”
- “Required or Mandatory MISRA C:2012 Rules Supported by Polyspace Bug Finder”
- “Checkers Deactivated in Polyspace as You Code Analysis”

Required MISRA C++:2008 Coding Rules Supported by Polyspace Bug Finder

The MISRA C++:2008 standard classifies the rules that compliant C++ code must follow as **Required**. Polyspace Bug Finder supports 195 out of 198 required MISRA C++:2008 coding rules.

Supported Rules

Polyspace supports these **Required** rules.

MISRA C++:2008 Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 0-1-1	A project shall not contain unreachable code	`MISRA C++:2008 Rule 0-1-1`
MISRA C++:2008 Rule 0-1-10	Every defined function shall be called at least once	`MISRA C++:2008 Rule 0-1-10`
MISRA C++:2008 Rule 0-1-11	There shall be no unused parameters (named or unnamed) in nonvirtual functions	`MISRA C++:2008 Rule 0-1-11`
MISRA C++:2008 Rule 0-1-12	There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it	`MISRA C++:2008 Rule 0-1-12`
MISRA C++:2008 Rule 0-1-2	A project shall not contain infeasible paths	`MISRA C++:2008 Rule 0-1-2`
MISRA C++:2008 Rule 0-1-3	A project shall not contain unused variables	`MISRA C++:2008 Rule 0-1-3`
MISRA C++:2008 Rule 0-1-4	A project shall not contain non-volatile POD variables having only one use	`MISRA C++:2008 Rule 0-1-4`
MISRA C++:2008 Rule 0-1-5	A project shall not contain unused type declarations	`MISRA C++:2008 Rule 0-1-5`
MISRA C++:2008 Rule 0-1-6	A project shall not contain instances of non-volatile variables being given values that are never subsequently used.	`MISRA C++:2008 Rule 0-1-6`
MISRA C++:2008 Rule 0-1-7	The value returned by a function having a non- void return type that is not an overloaded operator shall always be used	`MISRA C++:2008 Rule 0-1-7`
MISRA C++:2008 Rule 0-1-8	All functions with void return type shall have external side effect(s)	`MISRA C++:2008 Rule 0-1-8`

MISRA C++:2008 Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 0-1-9	There shall be no dead code	MISRA C++:2008 Rule 0-1-9
MISRA C++:2008 Rule 0-2-1	An object shall not be assigned to an overlapping object	MISRA C++:2008 Rule 0-2-1
MISRA C++:2008 Rule 0-3-2	If a function generates error information, then that error information shall be tested	MISRA C++:2008 Rule 0-3-2
MISRA C++:2008 Rule 10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy	MISRA C++:2008 Rule 10-1-2
MISRA C++:2008 Rule 10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy	MISRA C++:2008 Rule 10-1-3
MISRA C++:2008 Rule 10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy	MISRA C++:2008 Rule 10-3-1
MISRA C++:2008 Rule 10-3-2	Each overriding virtual function shall be declared with the virtual keyword	MISRA C++:2008 Rule 10-3-2
MISRA C++:2008 Rule 10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual	MISRA C++:2008 Rule 10-3-3
MISRA C++:2008 Rule 11-0-1	Member data in non- POD class types shall be private	MISRA C++:2008 Rule 11-0-1
MISRA C++:2008 Rule 12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor	MISRA C++:2008 Rule 12-1-1
MISRA C++:2008 Rule 12-1-3	All constructors that are callable with a single argument of fundamental type shall be declared explicit	MISRA C++:2008 Rule 12-1-3
MISRA C++:2008 Rule 12-8-1	A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member	MISRA C++:2008 Rule 12-8-1
MISRA C++:2008 Rule 12-8-2	The copy assignment operator shall be declared protected or private in an abstract class	MISRA C++:2008 Rule 12-8-2
MISRA C++:2008 Rule 14-5-1	A non-member generic function shall only be declared in a namespace that is not an associated namespace	MISRA C++:2008 Rule 14-5-1

MISRA C++:2008 Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 14-5-2	A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter	`MISRA C++:2008 Rule 14-5-2`
MISRA C++:2008 Rule 14-5-3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter	`MISRA C++:2008 Rule 14-5-3`
MISRA C++:2008 Rule 14-6-1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->	`MISRA C++:2008 Rule 14-6-1`
MISRA C++:2008 Rule 14-6-2	The function chosen by overload resolution shall resolve to a function declared previously in the translation unit	`MISRA C++:2008 Rule 14-6-2`
MISRA C++:2008 Rule 14-7-3	All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template	`MISRA C++:2008 Rule 14-7-3`
MISRA C++:2008 Rule 14-8-1	Overloaded function templates shall not be explicitly specialized	`MISRA C++:2008 Rule 14-8-1`
MISRA C++:2008 Rule 15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement	`MISRA C++:2008 Rule 15-0-3`
MISRA C++:2008 Rule 15-1-1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown	`MISRA C++:2008 Rule 15-1-1`
MISRA C++:2008 Rule 15-1-2	NULL shall not be thrown explicitly	`MISRA C++:2008 Rule 15-1-2`
MISRA C++:2008 Rule 15-1-3	An empty throw (throw;) shall only be used in the compound-statement of a catch handler	`MISRA C++:2008 Rule 15-1-3`
MISRA C++:2008 Rule 15-3-1	Exceptions shall be raised only after start-up and before termination of the program	`MISRA C++:2008 Rule 15-3-1`

MISRA C++:2008 Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases	MISRA C++:2008 Rule 15-3-3
MISRA C++:2008 Rule 15-3-4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point	MISRA C++:2008 Rule 15-3-4
MISRA C++:2008 Rule 15-3-5	A class type exception shall always be caught by reference	MISRA C++:2008 Rule 15-3-5
MISRA C++:2008 Rule 15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class	MISRA C++:2008 Rule 15-3-6
MISRA C++:2008 Rule 15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last	MISRA C++:2008 Rule 15-3-7
MISRA C++:2008 Rule 15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids	MISRA C++:2008 Rule 15-4-1
MISRA C++:2008 Rule 15-5-1	A class destructor shall not exit with an exception	MISRA C++:2008 Rule 15-5-1
MISRA C++:2008 Rule 15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s)	MISRA C++:2008 Rule 15-5-2
MISRA C++:2008 Rule 15-5-3	The terminate() function shall not be called implicitly	MISRA C++:2008 Rule 15-5-3
MISRA C++:2008 Rule 16-0-1	#include directives in a file shall only be preceded by other preprocessor directives or comments	MISRA C++:2008 Rule 16-0-1

MISRA C++:2008 Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 16-0-2	Macros shall only be #define 'd or #undef 'd in the global namespace	`MISRA C++:2008 Rule 16-0-2`
MISRA C++:2008 Rule 16-0-3	#undef shall not be used	`MISRA C++:2008 Rule 16-0-3`
MISRA C++:2008 Rule 16-0-4	Function-like macros shall not be defined	`MISRA C++:2008 Rule 16-0-4`
MISRA C++:2008 Rule 16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives	`MISRA C++:2008 Rule 16-0-5`
MISRA C++:2008 Rule 16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##	`MISRA C++:2008 Rule 16-0-6`
MISRA C++:2008 Rule 16-0-7	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator	`MISRA C++:2008 Rule 16-0-7`
MISRA C++:2008 Rule 16-0-8	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token	`MISRA C++:2008 Rule 16-0-8`
MISRA C++:2008 Rule 16-1-1	The defined preprocessor operator shall only be used in one of the two standard forms	`MISRA C++:2008 Rule 16-1-1`
MISRA C++:2008 Rule 16-1-2	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related	`MISRA C++:2008 Rule 16-1-2`
MISRA C++:2008 Rule 16-2-1	The preprocessor shall only be used for file inclusion and include guards	`MISRA C++:2008 Rule 16-2-1`
MISRA C++:2008 Rule 16-2-2	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers	`MISRA C++:2008 Rule 16-2-2`
MISRA C++:2008 Rule 16-2-3	Include guards shall be provided	`MISRA C++:2008 Rule 16-2-3`
MISRA C++:2008 Rule 16-2-4	The ', ", /* or // characters shall not occur in a header file name	`MISRA C++:2008 Rule 16-2-4`

MISRA C++:2008 Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 16-2-6	The #include directive shall be followed by either a <filename> or "filename" sequence	MISRA C++:2008 Rule 16-2-6
MISRA C++:2008 Rule 16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition	MISRA C++:2008 Rule 16-3-1
MISRA C++:2008 Rule 17-0-1	Reserved identifiers, macros and functions in the Standard Library shall not be defined, redefined or undefined	MISRA C++:2008 Rule 17-0-1
MISRA C++:2008 Rule 17-0-2	The names of standard library macros and objects shall not be reused	MISRA C++:2008 Rule 17-0-2
MISRA C++:2008 Rule 17-0-3	The names of standard library functions shall not be overridden	MISRA C++:2008 Rule 17-0-3
MISRA C++:2008 Rule 17-0-5	The setjmp macro and the longjmp function shall not be used	MISRA C++:2008 Rule 17-0-5
MISRA C++:2008 Rule 18-0-1	The C library shall not be used	MISRA C++:2008 Rule 18-0-1
MISRA C++:2008 Rule 18-0-2	The library functions atof, atoi and atol from library <cstdlib> shall not be used	MISRA C++:2008 Rule 18-0-2
MISRA C++:2008 Rule 18-0-3	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used	MISRA C++:2008 Rule 18-0-3
MISRA C++:2008 Rule 18-0-4	The time handling functions of library <ctime> shall not be used	MISRA C++:2008 Rule 18-0-4
MISRA C++:2008 Rule 18-0-5	The unbounded functions of library <cstring> shall not be used	MISRA C++:2008 Rule 18-0-5
MISRA C++:2008 Rule 18-2-1	The macro offsetof shall not be used	MISRA C++:2008 Rule 18-2-1
MISRA C++:2008 Rule 18-4-1	Dynamic heap memory allocation shall not be used	MISRA C++:2008 Rule 18-4-1
MISRA C++:2008 Rule 18-7-1	The signal handling facilities of <csignal> shall not be used	MISRA C++:2008 Rule 18-7-1
MISRA C++:2008 Rule 19-3-1	The error indicator errno shall not be used	MISRA C++:2008 Rule 19-3-1

MISRA C++:2008 Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 1-0-1	All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1"	`MISRA C++:2008 Rule 1-0-1`
MISRA C++:2008 Rule 27-0-1	The stream input/output library <cstdio> shall not be used	`MISRA C++:2008 Rule 27-0-1`
MISRA C++:2008 Rule 2-10-1	Different identifiers shall be typographically unambiguous	`MISRA C++:2008 Rule 2-10-1`
MISRA C++:2008 Rule 2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope	`MISRA C++:2008 Rule 2-10-2`
MISRA C++:2008 Rule 2-10-3	A typedef name (including qualification, if any) shall be a unique identifier	`MISRA C++:2008 Rule 2-10-3`
MISRA C++:2008 Rule 2-10-4	A class, union or enum name (including qualification, if any) shall be a unique identifier	`MISRA C++:2008 Rule 2-10-4`
MISRA C++:2008 Rule 2-10-6	If an identifier refers to a type, it shall not also refer to an object or a function in the same scope	`MISRA C++:2008 Rule 2-10-6`
MISRA C++:2008 Rule 2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used	`MISRA C++:2008 Rule 2-13-1`
MISRA C++:2008 Rule 2-13-2	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used	`MISRA C++:2008 Rule 2-13-2`
MISRA C++:2008 Rule 2-13-3	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type	`MISRA C++:2008 Rule 2-13-3`
MISRA C++:2008 Rule 2-13-4	Literal suffixes shall be upper case	`MISRA C++:2008 Rule 2-13-4`
MISRA C++:2008 Rule 2-13-5	Narrow and wide string literals shall not be concatenated	`MISRA C++:2008 Rule 2-13-5`
MISRA C++:2008 Rule 2-3-1	Trigraphs shall not be used	`MISRA C++:2008 Rule 2-3-1`
MISRA C++:2008 Rule 2-7-1	The character sequence /* shall not be used within a C-style comment	`MISRA C++:2008 Rule 2-7-1`
MISRA C++:2008 Rule 2-7-2	Sections of code shall not be "commented out" using C-style comments	`MISRA C++:2008 Rule 2-7-2`

MISRA C++:2008 Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 3-1-1	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule	`MISRA C++:2008 Rule 3-1-1`
MISRA C++:2008 Rule 3-1-2	Functions shall not be declared at block scope	`MISRA C++:2008 Rule 3-1-2`
MISRA C++:2008 Rule 3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization	`MISRA C++:2008 Rule 3-1-3`
MISRA C++:2008 Rule 3-2-1	All declarations of an object or function shall have compatible types	`MISRA C++:2008 Rule 3-2-1`
MISRA C++:2008 Rule 3-2-2	The One Definition Rule shall not be violated	`MISRA C++:2008 Rule 3-2-2`
MISRA C++:2008 Rule 3-2-3	A type, object or function that is used in multiple translation units shall be declared in one and only one file	`MISRA C++:2008 Rule 3-2-3`
MISRA C++:2008 Rule 3-2-4	An identifier with external linkage shall have exactly one definition	`MISRA C++:2008 Rule 3-2-4`
MISRA C++:2008 Rule 3-3-1	Objects or functions with external linkage shall be declared in a header file	`MISRA C++:2008 Rule 3-3-1`
MISRA C++:2008 Rule 3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier	`MISRA C++:2008 Rule 3-3-2`
MISRA C++:2008 Rule 3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility	`MISRA C++:2008 Rule 3-4-1`
MISRA C++:2008 Rule 3-9-1	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations	`MISRA C++:2008 Rule 3-9-1`
MISRA C++:2008 Rule 3-9-3	The underlying bit representations of floating-point values shall not be used	`MISRA C++:2008 Rule 3-9-3`
MISRA C++:2008 Rule 4-10-1	NULL shall not be used as an integer value	`MISRA C++:2008 Rule 4-10-1`

MISRA C++:2008 Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 4-10-2	Literal zero (0) shall not be used as the null-pointer-constant	MISRA C++:2008 Rule 4-10-2
MISRA C++:2008 Rule 4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator	MISRA C++:2008 Rule 4-5-1
MISRA C++:2008 Rule 4-5-2	Expressions with type enum shall not be used as operands to built- in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=	MISRA C++:2008 Rule 4-5-2
MISRA C++:2008 Rule 4-5-3	Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator N	MISRA C++:2008 Rule 4-5-3
MISRA C++:2008 Rule 5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits	MISRA C++:2008 Rule 5-0-1
MISRA C++:2008 Rule 5-0-10	If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand	MISRA C++:2008 Rule 5-0-10
MISRA C++:2008 Rule 5-0-11	The plain char type shall only be used for the storage and use of character values	MISRA C++:2008 Rule 5-0-11
MISRA C++:2008 Rule 5-0-12	Signed char and unsigned char type shall only be used for the storage and use of numeric values	MISRA C++:2008 Rule 5-0-12

MISRA C++:2008 Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 5-0-13	The condition of an if-statement and the condition of an iteration- statement shall have type bool	MISRA C++:2008 Rule 5-0-13
MISRA C++:2008 Rule 5-0-14	The first operand of a conditional-operator shall have type bool	MISRA C++:2008 Rule 5-0-14
MISRA C++:2008 Rule 5-0-15	Array indexing shall be the only form of pointer arithmetic	MISRA C++:2008 Rule 5-0-15
MISRA C++:2008 Rule 5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	MISRA C++:2008 Rule 5-0-16
MISRA C++:2008 Rule 5-0-17	Subtraction between pointers shall only be applied to pointers that address elements of the same array	MISRA C++:2008 Rule 5-0-17
MISRA C++:2008 Rule 5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array	MISRA C++:2008 Rule 5-0-18
MISRA C++:2008 Rule 5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection	MISRA C++:2008 Rule 5-0-19
MISRA C++:2008 Rule 5-0-20	Non-constant operands to a binary bitwise operator shall have the same underlying type	MISRA C++:2008 Rule 5-0-20
MISRA C++:2008 Rule 5-0-21	Bitwise operators shall only be applied to operands of unsigned underlying type	MISRA C++:2008 Rule 5-0-21
MISRA C++:2008 Rule 5-0-3	A cvalue expression shall not be implicitly converted to a different underlying type	MISRA C++:2008 Rule 5-0-3
MISRA C++:2008 Rule 5-0-4	An implicit integral conversion shall not change the signedness of the underlying type	MISRA C++:2008 Rule 5-0-4
MISRA C++:2008 Rule 5-0-5	There shall be no implicit floating-integral conversions	MISRA C++:2008 Rule 5-0-5
MISRA C++:2008 Rule 5-0-6	An implicit integral or floating-point conversion shall not reduce the size of the underlying type	MISRA C++:2008 Rule 5-0-6

MISRA C++:2008 Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression	MISRA C++:2008 Rule 5-0-7
MISRA C++:2008 Rule 5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression	MISRA C++:2008 Rule 5-0-8
MISRA C++:2008 Rule 5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression	MISRA C++:2008 Rule 5-0-9
MISRA C++:2008 Rule 5-14-1	The right hand operand of a logical && or || operator shall not contain side effects	MISRA C++:2008 Rule 5-14-1
MISRA C++:2008 Rule 5-18-1	The comma operator shall not be used	MISRA C++:2008 Rule 5-18-1
MISRA C++:2008 Rule 5-2-1	Each operand of a logical && or || shall be a postfix-expression	MISRA C++:2008 Rule 5-2-1
MISRA C++:2008 Rule 5-2-11	The comma operator, && operator and the || operator shall not be overloaded	MISRA C++:2008 Rule 5-2-11
MISRA C++:2008 Rule 5-2-12	An identifier with array type passed as a function argument shall not decay to a pointer	MISRA C++:2008 Rule 5-2-12
MISRA C++:2008 Rule 5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of `dynamic_cast`	MISRA C++:2008 Rule 5-2-2
MISRA C++:2008 Rule 5-2-4	C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used	MISRA C++:2008 Rule 5-2-4
MISRA C++:2008 Rule 5-2-5	A cast shall not remove any const or volatile qualification from the type of a pointer or reference	MISRA C++:2008 Rule 5-2-5
MISRA C++:2008 Rule 5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type	MISRA C++:2008 Rule 5-2-6

MISRA C++:2008 Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 5-2-7	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly	`MISRA C++:2008 Rule 5-2-7`
MISRA C++:2008 Rule 5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type	`MISRA C++:2008 Rule 5-2-8`
MISRA C++:2008 Rule 5-3-1	Each operand of the ! operator, the logical && or the logical || operators shall have type bool	`MISRA C++:2008 Rule 5-3-1`
MISRA C++:2008 Rule 5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned	`MISRA C++:2008 Rule 5-3-2`
MISRA C++:2008 Rule 5-3-3	The unary & operator shall not be overloaded	`MISRA C++:2008 Rule 5-3-3`
MISRA C++:2008 Rule 5-3-4	Evaluation of the operand to the sizeof operator shall not contain side effects	`MISRA C++:2008 Rule 5-3-4`
MISRA C++:2008 Rule 5-8-1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand	`MISRA C++:2008 Rule 5-8-1`
MISRA C++:2008 Rule 6-2-1	Assignment operators shall not be used in sub-expressions	`MISRA C++:2008 Rule 6-2-1`
MISRA C++:2008 Rule 6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality	`MISRA C++:2008 Rule 6-2-2`
MISRA C++:2008 Rule 6-2-3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character	`MISRA C++:2008 Rule 6-2-3`
MISRA C++:2008 Rule 6-3-1	The statement forming the body of a switch, while, do while or for statement shall be a compound statement	`MISRA C++:2008 Rule 6-3-1`

MISRA C++:2008 Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 6-4-1	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement	`MISRA C++:2008 Rule 6-4-1`
MISRA C++:2008 Rule 6-4-2	All if â€¦ else if constructs shall be terminated with an else clause	`MISRA C++:2008 Rule 6-4-2`
MISRA C++:2008 Rule 6-4-3	A switch statement shall be a well-formed switch statement	`MISRA C++:2008 Rule 6-4-3`
MISRA C++:2008 Rule 6-4-4	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	`MISRA C++:2008 Rule 6-4-4`
MISRA C++:2008 Rule 6-4-5	An unconditional throw or break statement shall terminate every non - empty switch-clause	`MISRA C++:2008 Rule 6-4-5`
MISRA C++:2008 Rule 6-4-6	The final clause of a `switch` statement shall be the default-clause	`MISRA C++:2008 Rule 6-4-6`
MISRA C++:2008 Rule 6-4-7	The condition of a switch statement shall not have bool type	`MISRA C++:2008 Rule 6-4-7`
MISRA C++:2008 Rule 6-4-8	Every switch statement shall have at least one case-clause	`MISRA C++:2008 Rule 6-4-8`
MISRA C++:2008 Rule 6-5-1	A for loop shall contain a single loop-counter which shall not have floating type	`MISRA C++:2008 Rule 6-5-1`
MISRA C++:2008 Rule 6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=	`MISRA C++:2008 Rule 6-5-2`
MISRA C++:2008 Rule 6-5-3	The loop-counter shall not be modified within condition or statement	`MISRA C++:2008 Rule 6-5-3`
MISRA C++:2008 Rule 6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop	`MISRA C++:2008 Rule 6-5-4`

MISRA C++:2008 Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 6-5-5	A loop-control-variable other than the loop-counter shall not be modified within condition or expression	`MISRA C++:2008 Rule 6-5-5`
MISRA C++:2008 Rule 6-5-6	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool	`MISRA C++:2008 Rule 6-5-6`
MISRA C++:2008 Rule 6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement	`MISRA C++:2008 Rule 6-6-1`
MISRA C++:2008 Rule 6-6-2	The goto statement shall jump to a label declared later in the same function body	`MISRA C++:2008 Rule 6-6-2`
MISRA C++:2008 Rule 6-6-3	The `continue` statement shall only be used within a well-formed `for` loop	`MISRA C++:2008 Rule 6-6-3`
MISRA C++:2008 Rule 6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination	`MISRA C++:2008 Rule 6-6-4`
MISRA C++:2008 Rule 6-6-5	A function shall have a single point of exit at the end of the function	`MISRA C++:2008 Rule 6-6-5`
MISRA C++:2008 Rule 7-1-1	A variable which is not modified shall be const qualified	`MISRA C++:2008 Rule 7-1-1`
MISRA C++:2008 Rule 7-1-2	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified	`MISRA C++:2008 Rule 7-1-2`
MISRA C++:2008 Rule 7-2-1	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	`MISRA C++:2008 Rule 7-2-1`
MISRA C++:2008 Rule 7-3-1	The global namespace shall only contain main, namespace declarations and extern "C" declarations	`MISRA C++:2008 Rule 7-3-1`
MISRA C++:2008 Rule 7-3-2	The identifier main shall not be used for a function other than the global function main	`MISRA C++:2008 Rule 7-3-2`

MISRA C++:2008 Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 7-3-3	There shall be no unnamed namespaces in header files	MISRA C++:2008 Rule 7-3-3
MISRA C++:2008 Rule 7-3-4	using-directives shall not be used	MISRA C++:2008 Rule 7-3-4
MISRA C++:2008 Rule 7-3-5	Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier	MISRA C++:2008 Rule 7-3-5
MISRA C++:2008 Rule 7-3-6	using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files	MISRA C++:2008 Rule 7-3-6
MISRA C++:2008 Rule 7-4-2	Assembler instructions shall only be introduced using the asm declaration	MISRA C++:2008 Rule 7-4-2
MISRA C++:2008 Rule 7-4-3	Assembly language shall be encapsulated and isolated	MISRA C++:2008 Rule 7-4-3
MISRA C++:2008 Rule 7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function	MISRA C++:2008 Rule 7-5-1
MISRA C++:2008 Rule 7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist	MISRA C++:2008 Rule 7-5-2
MISRA C++:2008 Rule 7-5-3	A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference	MISRA C++:2008 Rule 7-5-3
MISRA C++:2008 Rule 8-0-1	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively	MISRA C++:2008 Rule 8-0-1
MISRA C++:2008 Rule 8-3-1	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments	MISRA C++:2008 Rule 8-3-1
MISRA C++:2008 Rule 8-4-1	Functions shall not be defined using the ellipsis notation	MISRA C++:2008 Rule 8-4-1

MISRA C++:2008 Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 8-4-2	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration	`MISRA C++:2008 Rule 8-4-2`
MISRA C++:2008 Rule 8-4-3	All exit paths from a function with non- void return type shall have an explicit return statement with an expression	`MISRA C++:2008 Rule 8-4-3`
MISRA C++:2008 Rule 8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &	`MISRA C++:2008 Rule 8-4-4`
MISRA C++:2008 Rule 8-5-1	All variables shall have a defined value before they are used	`MISRA C++:2008 Rule 8-5-1`
MISRA C++:2008 Rule 8-5-2	Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures	`MISRA C++:2008 Rule 8-5-2`
MISRA C++:2008 Rule 8-5-3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized	`MISRA C++:2008 Rule 8-5-3`
MISRA C++:2008 Rule 9-3-1	const member functions shall not return non-const pointers or references to class-data	`MISRA C++:2008 Rule 9-3-1`
MISRA C++:2008 Rule 9-3-2	Member functions shall not return non-const handles to class-data	`MISRA C++:2008 Rule 9-3-2`
MISRA C++:2008 Rule 9-3-3	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const	`MISRA C++:2008 Rule 9-3-3`
MISRA C++:2008 Rule 9-5-1	Unions shall not be used	`MISRA C++:2008 Rule 9-5-1`
MISRA C++:2008 Rule 9-6-2	Bit-fields shall be either bool type or an explicitly unsigned or signed integral type	`MISRA C++:2008 Rule 9-6-2`
MISRA C++:2008 Rule 9-6-3	Bit-fields shall not have enum type	`MISRA C++:2008 Rule 9-6-3`
MISRA C++:2008 Rule 9-6-4	Named bit-fields with signed integer type shall have a length of more than one bit	`MISRA C++:2008 Rule 9-6-4`

Unsupported Rules

Polyspace does not supports these **Required** rules:

Rule	Description
5-17-1	The semantic equivalence between a binary operator and its assignment operator form shall be preserved.
14-7-1	All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.
14-7-2	For any given template specialization, an explicit instantiation of the template with the template-arguments used in the specialization shall not render the program ill-formed.

See Also

`Check MISRA C++:2008 (-misra-cpp)`

More About

- “Check for and Review Coding Standard Violations”
- “Coding Standards”
- “Required AUTOSAR C++14 Coding Rules Supported by Polyspace Bug Finder”
- “Statically Enforceable AUTOSAR C++14 Rules Supported by Polyspace Bug Finder”
- “Checkers Deactivated in Polyspace as You Code Analysis”

JSF AV C++ Coding Rules

Note Starting in a future release, Code Prover will not support checking compliance with external coding standards and calculating code metrics. Migrate to Bug Finder for these workflows. See “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.
19	<locale.h> and the setlocale function **shall not** be used.	setlocale and localeconv should not be used as a macro or a global with external "C" linkage.
20	The setjmp macro and the longjmp function **shall not** be used.	setjmp and longjmp 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.	signal and raise 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 atof, atoi and atol from library <stdlib.h> **shall not** be used.	atof, atoi and atol should not be used as a macro or a global with external "C" linkage.
24	The library functions abort, exit, getenv and system from library <stdlib.h> **shall not** be used.	abort, exit, getenv and system 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.	clock, difftime, mktime, asctime, ctime, gmtime, localtime and strftime 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: #ifndef, #define, #endif, #include.	
27	#ifndef, #define and #endif **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 #if !defined, #pragma once, #ifdef, and missing #define.
28	The #ifndef and #endif 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 #ifndef.

N.	JSF++ Definition	Polyspace Implementation
29	The `#define` preprocessor directive **shall not** be used to create inline macros. Inline functions shall be used instead.	Rule is split into two parts: the definition of a macro function (29.def) and the call of a macrofunction (29.use). Messages in report file: • 29.1 : The `#define` preprocessor directive shall not be used to create inline macros. • 29.2 : Inline functions shall be used instead of inline macros.
30	The `#define` preprocessor directive **shall not** be used to define constant values. Instead, the `const` qualifier **shall** be applied to variable declarations to specify constant values.	Reports `#define` of simple constants.
31	The `#define` preprocessor directive **will** only be used as part of the technique to prevent multiple inclusions of the same header file.	Detects use of `#define` that are not used to guard for multiple inclusion, assuming that rules 35 and 27 are not violated.
32	The `#include` preprocessor directive **will** only be used to include header (*.h) files.	

Header Files

N.	JSF++ Definition	Polyspace Implementation
33	The `#include` directive **shall** use the `<filename.h>` notation to include header files.	
35	A header file **will** contain a mechanism that prevents multiple inclusions of itself.	
39	Header files (`*.h`) **will not** contain non-const variable definitions or function definitions.	Reports definitions of global variables / function in header.

Style

N.	JSF++ Definition	Polyspace Implementation
40	Every implementation file shall include the header files that uniquely define the inline functions, types, and templates used.	Reports when type, template, or inline function is defined in source file.
41	Source lines **will** be kept to a length of 120 characters or less.	Polyspace ignores the newline character (`\n`) when counting the line length.
42	Each expression-statement **will** be on a separate line.	Reports when two consecutive expression statements are on the same line (unless the statements are part of a macro definition).
43	Tabs **should** be avoided.	

N.	JSF++ Definition	Polyspace Implementation
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.	*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis"*.
47	Identifiers **will not** begin with the underscore character '_'.	
48	Identifiers **will not** differ by: • 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: • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) only differ by the presence/absence of the underscore character. • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) only differ by a mixture of case. • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) only differ by letter `O`, 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: • The first word of the name of a class will begin with an uppercase letter. • The first word of the namespace of a class will begin with an uppercase letter.
51	All letters contained in function and variables names **will** be composed entirely of lowercase letters.	Messages in report file: • 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: • Identifier for enumerator value shall be lowercase. • Identifier for template constant parameter shall be lowercase.

N.	JSF++ Definition	Polyspace Implementation
53	Header files **will** always have file name extension of "`.h`".	`.H` is allowed if you set the option `-dos`.
53.1	The following character sequences **shall** not appear in header file names: `'`, `\`, `/*`, `//`, or `"`.	
54	Implementation files **will** always have a file name extension of ".cpp".	Not case sensitive if you set the option `-dos`.
57	The public, protected, and private sections of a class **will** be declared in that order.	
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: • 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.

N.	JSF++ Definition	Polyspace Implementation
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: • . • -> • ! • ~ • - • ++ • — 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: • `no copy constructor and no copy assign` • `no copy constructor` • `no copy assign`
77.1	The definition of a member function **shall not** contain default arguments that produce a signature identical to that of the implicitly-declared copy constructor for the corresponding class/structure.	Does not report when an explicit copy constructor exists.

<table>
<tr><th>N.</th><th>JSF++ Definition</th><th>Polyspace Implementation</th></tr>
<tr><td>78</td><td>All base classes with a virtual function shall define a virtual destructor.</td><td></td></tr>
<tr><td>79</td><td>All resources acquired by a class shall be released by the class’s destructor.</td><td>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”.</td></tr>
<tr><td>81</td><td>The assignment operator shall handle self-assignment correctly</td><td>Reports when copy assignment body does not begin with “<code>if (this != arg)</code>”

A violation is not raised if an empty <code>else</code> statement follows the <code>if</code>, or the body contains only a return statement.

A violation is raised when the <code>if</code> statement is followed by a statement other than the return statement.</td></tr>
<tr><td>82</td><td>An assignment operator shall return a reference to <code>*this</code>.</td><td>The following operators should return <code>*this</code> on method, and <code>*first_arg</code> on plain function:

• <code>operator=</code>
• <code>operator+=</code>
• <code>operator-=</code>
• <code>operator*=</code>
• <code>operator >>=</code>
• <code>operator <<=</code>
• <code>operator /=</code>
• <code>operator %=</code>
• <code>operator |=</code>
• <code>operator &=</code>
• <code>operator ^=</code>
• Prefix <code>operator++</code>
• Prefix <code>operator--</code>

Does not report when no return exists.

No special message if type does not match.

Messages in report file:

• An assignment operator shall return a reference to <code>*this</code>.
• An assignment operator shall return a reference to its first arg.</td></tr>
</table>

N.	JSF++ Definition	Polyspace Implementation
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.
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: • Multiple inheritance on public implementation shall not be allowed: *`<public_base_class>`* is not an interface. • Multiple inheritance on protected implementation shall not be allowed : *`<protected_base_class_1>`*. • *`<protected_base_class_2>`* 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 `main()`, **should** be placed in some namespace.	
99	Namespaces **will not** be nested more than two levels deep.	

Templates

N.	JSF++ Definition	Polyspace Implementation
104	A template specialization **shall** be declared before its use.	Reports the actual compilation error message.

Functions

N.	JSF++ Definition	Polyspace Implementation
107	Functions **shall** always be declared at file scope.	
108	Functions with variable numbers of arguments **shall not** be used.	
109	A function definition should not be placed in a class specification unless the function is intended to be inlined.	Reports when "inline" is not in the definition of a member function inside the class definition.
110	Functions with more than 7 arguments **will not** be used.	
111	A function **shall not** return a pointer or reference to a non-static local object.	Simple cases without alias effect detected.
113	Functions **will** have a single exit point.	Reports first return, or once per function.
114	All exit points of value-returning functions **shall** be through return statements.	
116	Small, concrete-type arguments (two or three words in size) **should** be passed by value if changes made to formal parameters should not be reflected in the calling function.	Report constant parameters references with `sizeof <= 2 * sizeof(int)`. Does not report for copy-constructor.
117	Arguments **should** be passed by reference if NULL values are not possible: • 117.1: An object should be passed as `const T&` if the function should not change the value of the object. • 117.2: An object should be passed as `T&` if the function may change the value of the object.	The checker flags a parameter passed by pointer if the parameter is not compared against `NULL` or `nullptr` 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. The checker does not raise a violation: • If a parameter is passed using a smart pointer. Only raw pointers are considered. • If the pointer parameter is not dereferenced within the function.

N.	JSF++ Definition	Polyspace Implementation
119	Functions **shall** not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).	The checker reports each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported. You can calculate the total number of recursion cycles using the code complexity metric `Number of Recursions`. Note that unlike the checker, the metric also considers implicit calls, for instance, to compiler-generated constructors during object creation.
121	Only functions with 1 or 2 statements **should** be considered candidates for inline functions.	Reports inline functions with more than 2 statements.
122	Trivial accessor and mutator functions **should** be inlined.	The checker uses the following criteria to determine if a method is trivial: • An accessor method is trivial if it has no parameters and contains one `return` statement that returns a non-static data member or a reference to a non-static data member. The return type of the method must exactly match or be a reference to the type of the data member. • A mutator method is trivial if it has a `void` return type, one parameter and contains one assignment statement that assigns the parameter to a non-static data member. The parameter type must exactly match or be a reference to the type of the data member. The checker reports trivial accessor and mutator methods defined outside their classes without the `inline` keyword. The checker does not flag template methods or virtual methods.

Comments

N.	JSF++ Definition	Polyspace Implementation
126	Only valid C++ style comments (`//`) **shall** be used.	

N.	JSF++ Definition	Polyspace Implementation

127	Code that is not used (commented out) shall be deleted.	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.
		
		The checker does not flag the following comments even if they contain code:
		• Doxygen comments beginning with `/**`, `/*!`, `///` or `//!`.
		• Comments that repeat the same symbol several times, for instance, the symbol = here:
		`// ================================`
		`// A comment`
		`// ================================*/`
		• Comments on the first line of a file.
		• Comments that mix the C style (`/* */`) and C++ style (`//`).
		
		The checker considers that these comments are meant for documentation purposes or entered deliberately with some forethought.

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).	Reports when a file does not begin with two comment lines. **Note**: This rule cannot be annotated in the source code.

Declarations and Definitions

N.	JSF++ Definition	Polyspace Implementation
135	Identifiers in an inner scope **shall not** use the same name as an identifier in an outer scope, and therefore hide that identifier.	

N.	JSF++ Definition	Polyspace Implementation
136	Declarations should be at the smallest feasible scope.	Reports when: • 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. **Note** • 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.	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 singe file. The checker is not raised on the declarations of objects that remain unused, such as: • Noninstantiated templates • Uncalled `static` or `extern` functions • Uncalled and undefined local functions • Unused types and variables *This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis"*.
138	Identifiers **shall not** simultaneously have both internal and external linkage in the same translation unit.	
139	External objects will not be declared in more than one file.	Reports all duplicate declarations inside a translation unit. Reports when the declaration localization is not the same in all translation units.
140	The register storage class specifier **shall not** be used.	
141	A class, structure, or enumeration **will not** be declared in the definition of its type.	

Initialization

N.	JSF++ Definition	Polyspace Implementation
142	All variables **shall** be initialized before use.	Polyspace reports a violation of this rule if your code contains these issues: • `Non-initialized variable` • `Member not initialized in constructor` • `Non-initialized pointer`
144	Braces **shall** be used to indicate and match the structure in the non-zero initialization of arrays and structures.	This covers partial initialization.
145	In an enumerator list, the '=' construct **shall not** be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	Generates one report for an enumerator list.

Types

N.	JSF++ Definition	Polyspace Implementation
147	The underlying bit representations of floating point numbers **shall not** be used in any way by the programmer.	Reports on casts with float pointers (except with `void*`).
148	Enumeration types shall be used instead of integer types (and constants) to select from a limited series of choices.	Reports when non enumeration types are used in switches.

Constants

N.	JSF++ Definition	Polyspace Implementation
149	Octal constants (other than zero) **shall not** be used.	
150	Hexadecimal constants **will** be represented using all uppercase letters.	
151	Numeric values in code **will not** be used; symbolic values will be used instead.	Reports direct numeric constants (except integer/float value `1, 0`) in expressions, non `-const` initializations. and switch cases. char constants are allowed. Does not report on templates non-type parameter.
151.1	A string literal shall not be modified.	The rule checker flags assignment of string literals to: • 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

<table>
<tr><th>N.</th><th>JSF++ Definition</th><th>Polyspace Implementation</th></tr>
<tr><td>157</td><td>The right hand operand of a && or || operator shall not contain side effects.</td><td>Assumes rule 159 is not violated.
Messages in report file:
• The right hand operand of a && operator shall not contain side effects.
• The right hand operand of a || operator shall not contain side effects.</td></tr>
<tr><td>158</td><td>The operands of a logical && or || shall be parenthesized if the operands contain binary operators.</td><td>Messages in report file:
• 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: <code>X || Y || Z , Z&&Y &&Z</code></td></tr>
<tr><td>159</td><td>Operators ||, &&, and unary & shall not be overloaded.</td><td>Messages in report file:
• Unary operator & shall not be overloaded.
• Operator || shall not be overloaded.
• Operator && shall not be overloaded.</td></tr>
<tr><td>160</td><td>An assignment expression shall be used only as the expression in an expression statement.</td><td>Only simple assignment, not +=, ++, etc.</td></tr>
<tr><td>162</td><td>Signed and unsigned values shall not be mixed in arithmetic or comparison operations.</td><td></td></tr>
<tr><td>163</td><td>Unsigned arithmetic shall not be used.</td><td></td></tr>
</table>

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: • the same object, • the same function, • members of the same object, or • elements of the same array (including one past the end of the same array).	Reports when relational operator are used on pointer types (casts ignored).
173	The address of an object with automatic storage **shall not** be assigned to an object which persists after the object has ceased to exist.	
174	The null pointer **shall not** be de-referenced.	Done with checks in software.
175	A pointer **shall not** be compared to `NULL` or be assigned `NULL`; use plain `0` instead.	Reports usage of `NULL` macro in pointer contexts.
176	A `typedef` **will** be used to simplify program syntax when declaring function pointers.	Reports non-`typedef` function pointers, or pointers to member functions for types of variables, fields, parameters. Returns type of function, cast, and exception specification.

Type Conversions

N.	JSF++ Definition	Polyspace Implementation
177	User-defined conversion functions **should** be avoided.	Reports user defined conversion function, non-explicit constructor with one parameter or default value for others (even undefined ones). Does not report copy-constructor. Additional message for constructor case: This constructor should be flagged as "explicit".
178	Down casting (casting from base to derived class) **shall** only be allowed through one of the following mechanism: • 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, dynamic_cast 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 dynamic_cast.
180	Implicit conversions that may result in a loss of information **shall not** be used.	Reports the following implicit casts : `integer => smaller integer unsigned => smaller or eq signed signed => smaller or eq un-signed integer => float float => integer` Does not report for cast to `bool` reports for implicit cast on constant done with the option `-scalar-overflows-checks signed-and-unsigned`
181	Redundant explicit casts **will not** be used.	Reports useless cast: `cast T to T`. Casts to equivalent `typedefs` are also reported.
182	Type casting from any type to or from pointers **shall not** be used.	Does not report when Rule 181 applies.
184	Floating point numbers **shall not** be converted to integers unless such a conversion is a specified algorithmic requirement or is necessary for a hardware interface.	Reports `float->int` conversions. Does not report implicit ones.
185	C++ style casts (`const_cast`, `reinterpret_cast`, and `static_cast`) **shall** be used instead of the traditional C-style casts.	

Flow Control Standards

N.	JSF++ Definition	Polyspace Implementation
186	There **shall** be no unreachable code.	Done with gray checks in the software.

N.	JSF++ Definition	Polyspace Implementation
187	All non-null statements **shall** potentially have a side-effect.	
188	Labels **will not** be used, except in switch statements.	
189	The `goto` statement **shall** not be used.	
190	The `continue` statement **shall not** be used.	
191	The `break` statement **shall not** be used (except to terminate the cases of a switch statement).	
192	All `if`, `else if` constructs will contain either a final `else` clause or a comment indicating why a final `else` clause is not necessary.	`else if` should contain an `else` clause.
193	Every non-empty `case` clause in a switch statement **shall** be terminated with a `break` statement.	
194	All `switch` statements that do not intend to test for every enumeration value **shall** contain a final `default` clause.	Reports only for missing `default`.
195	A `switch` expression **will** not represent a Boolean value.	
196	Every `switch` statement **will** have at least two cases and a potential `default`.	
197	Floating point variables **shall not** be used as loop counters.	Assumes 1 loop parameter.
198	The initialization expression in a `for` loop **will** perform no actions other than to initialize the value of a single `for` loop parameter.	Reports if `loop` parameter cannot be determined. Assumes Rule 200 is not violated. The `loop variable` parameter is assumed to be a variable.
199	The increment expression in a `for` 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 `for` loops **will not** be used; a `while` loop will be used instead.	
201	Numeric variables being used within a *for* loop for iteration counting shall not be modified in the body of the loop.	Assumes 1 loop parameter (AV rule 198), and no alias writes.

Expressions

N.	JSF++ Definition	Polyspace Implementation
202	Floating point variables **shall not** be tested for exact equality or inequality.	Reports only direct equality/inequality. Check done for all expressions.
203	Evaluation of expressions **shall not** lead to overflow/underflow.	Done with overflow checks in the software.

N.	JSF++ Definition	Polyspace Implementation
204	A single operation with side-effects shall only be used in the following contexts: • 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: • 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: • A variable is written and reused within the same expression. • A volatile variable is accessed more than once.
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 `int`, `short`, `long`, `float` and `double` **shall not** be used, but specific-length equivalents should be `typedef`'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.	Reports when a binary operation has one operand that is not parenthesized and is an operation with inferior precedence level. Reports bitwise and shifts operators that are used without parenthesis and binary operation arguments.

N.	JSF++ Definition	Polyspace Implementation
215	Pointer arithmetic **will not** be used.	Reports:`p + I` `p - I` `p++` `p--` `p+=` `p-=` Allows `p[i]`.

Unsupported JSF++ Rules

- “Code Size and Complexity” on page 17-115
- “Rules” on page 17-115
- “Environment” on page 17-116
- “Libraries” on page 17-116
- “Header Files” on page 17-116
- “Style” on page 17-116
- “Classes” on page 17-117
- “Namespaces” on page 17-118
- “Templates” on page 17-118
- “Functions” on page 17-118
- “Comments” on page 17-118
- “Initialization” on page 17-119
- “Types” on page 17-119
- “Unions and Bit Fields” on page 17-119
- “Operators” on page 17-119
- “Type Conversions” on page 17-119
- “Expressions” on page 17-120
- “Memory Allocation” on page 17-120
- “Portable Code” on page 17-120
- “Efficiency Considerations” on page 17-120
- “Miscellaneous” on page 17-120
- “Testing” on page 17-121

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: • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool)

N.	JSF++ Definition
5	To break a “will” or a “shall” rule, the following approvals must be received by the developer: • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool) • approval from the software product manager (obtained by the unit approval in the developmental CM tool)
6	Each deviation from a “shall” rule shall be documented in the file that contains the deviation. Deviations from this rule shall not be allowed, AV Rule 5 notwithstanding.
7	Approval will not be required for a deviation from a “shall” or “will” rule that complies with an exception specified by that rule.

Environment

N.	JSF++ Definition
10	Values of character types will be restricted to a defined and documented subset of ISO 10646 1.

Libraries

N.	JSF++ Definition
16	Only DO-178B level A [15] certifiable or SEAL 1 C/C++ libraries shall be used with safety-critical (i.e. SEAL 1) code.

Header Files

N.	JSF++ Definition
34	Header files should contain logically related declarations only.
36	Compilation dependencies should be minimized when possible.
37	Header (include) files should include only those header files that are required for them to successfully compile. Files that are only used by the associated .cpp file should be placed in the .cpp file — not the .h file.
38	Declarations of classes that are only accessed via pointers (*) or references (&) should be supplied by forward headers that contain only forward declarations.

Style

N.	JSF++ Definition
45	All words in an identifier will be separated by the ‘_’ character.
49	All acronyms in an identifier will be composed of uppercase letters.
55	The name of a header file should reflect the logical entity for which it provides declarations.
56	The name of an implementation file should reflect the logical entity for which it provides definitions and have a “.cpp” extension (this name will normally be identical to the header file that provides the corresponding declarations.) At times, more than one .cpp file for a given logical entity will be required. In these cases, a suffix should be appended to reflect a logical differentiation.

Classes

N.	JSF++ Definition
64	A class interface should be complete and minimal.
65	A structure should be used to model an entity that does not require an invariant.
66	A class should be used to model an entity that maintains an invariant.
69	A member function that does not affect the state of an object (its instance variables) will be declared const. Member functions should be const by default. Only when there is a clear, explicit reason should the const modifier on member functions be omitted.
70	A class will have friends only when a function or object requires access to the private elements of the class, but is unable to be a member of the class for logical or efficiency reasons.
70.1	An object shall not be improperly used before its lifetime begins or after its lifetime ends.
71	Calls to an externally visible operation of an object, other than its constructors, shall not be allowed until the object has been fully initialized.
72	The invariant for a class should be: • A part of the postcondition of every class constructor, • A part of the precondition of the class destructor (if any), • A part of the precondition and postcondition of every other publicly accessible operation.
73	Unnecessary default constructors shall not be defined.
77	A copy constructor shall copy all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).
80	The default copy and assignment operators will be used for classes when those operators offer reasonable semantics.
84	Operator overloading will be used sparingly and in a conventional manner.
85	When two operators are opposites (such as == and !=), both will be defined and one will be defined in terms of the other.
86	Concrete types should be used to represent simple independent concepts.
87	Hierarchies should be based on abstract classes.
90	Heavily used interfaces should be minimal, general and abstract.
91	Public inheritance will be used to implement “is-a” relationships.
92	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: • 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. 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.
93	“has-a” or “is-implemented-in-terms-of” relationships will be modeled through membership or non-public inheritance.

Namespaces

N.	JSF++ Definition
100	Elements from a namespace should be selected as follows: • using declaration or explicit qualification for few (approximately five) names, • using directive for many names.

Templates

N.	JSF++ Definition
101	Templates shall be reviewed as follows: 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	Arguments should be passed via pointers if NULL values are possible: • **118.1** – An object should be passed as `const T*` if its value should not be modified. • **118.2** – An object should be passed as `T*` if its value may be modified.
120	Overloaded operations or methods should form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal parameters.
123	The number of accessor and mutator functions should be minimized.
124	Trivial forwarding functions should be inlined.
125	Unnecessary temporary objects should be avoided.

Comments

N.	JSF++ Definition
128	Comments that document actions or sources (e.g. tables, figures, paragraphs, etc.) outside of the file being documented will not be allowed.
129	Comments in header files should describe the externally visible behavior of the functions or classes being documented.
130	The purpose of every line of executable code should be explained by a comment, although one comment may describe more than one line of code.

N.	JSF++ Definition
131	One should avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).
132	Each variable declaration, typedef, enumeration value, and structure member will be commented.
134	Assumptions (limitations) made by functions should be documented in the function’s preamble.

Initialization

N.	JSF++ Definition
143	Variables will not be introduced until they can be initialized with meaningful values. (See also AV Rule 136, AV Rule 142, and AV Rule 73 concerning declaration scope, initialization before use, and default constructors respectively.)

Types

N.	JSF++ Definition
146	Floating point implementations shall comply with a defined floating point standard. The standard that will be used is the ANSI/IEEE® Std 754 [1].

Unions and Bit Fields

N.	JSF++ Definition
155	Bit-fields will not be used to pack data into a word for the sole purpose of saving space.

Operators

N.	JSF++ Definition
167	The implementation of integer division in the chosen compiler shall be determined, documented and taken into account.

Type Conversions

N.	JSF++ Definition
183	Every possible measure should be taken to avoid type casting.

Expressions

N.	JSF++ Definition
204	A single operation with side-effects shall only be used in the following contexts: 1 by itself 2 the right-hand side of an assignment 3 a condition 4 the only argument expression with a side-effect in a function call 5 condition of a loop 6 switch condition 7 single part of a chained operation

Memory Allocation

N.	JSF++ Definition
207	Unencapsulated global data will be avoided.

Portable Code

N.	JSF++ Definition
210	Algorithms shall not make assumptions concerning how data is represented in memory (e.g. big endian vs. little endian, base class subobject ordering in derived classes, nonstatic data member ordering across access specifiers, etc.).
210.1	Algorithms shall not make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier.
211	Algorithms shall not assume that shorts, ints, longs, floats, doubles or long doubles begin at particular addresses.
212	Underflow or overflow functioning shall not be depended on in any special way.
214	Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.

Efficiency Considerations

N.	JSF++ Definition
216	Programmers should not attempt to prematurely optimize code.

Miscellaneous

N.	JSF++ Definition
217	Compile-time and link-time errors should be preferred over run-time errors.
218	Compiler warning levels will be set in compliance with project policies.

Testing

N.	JSF++ Definition
219	All tests applied to a base class interface shall be applied to all derived class interfaces as well. If the derived class poses stronger postconditions/invariants, then the new postconditions /invariants shall be substituted in the derived class tests.
220	Structural coverage algorithms shall be applied against flattened classes.
221	Structural coverage of a class within an inheritance hierarchy containing virtual functions shall include testing every possible resolution for each set of identical polymorphic references.

Required AUTOSAR C++14 Coding Rules Supported by Polyspace Bug Finder

The AUTOSAR C++14 standard classifies the rules that compliant C++ code must follow as **Required**. Polyspace Bug Finder supports 337 out of 362 **Required** AUTOSAR C++14 coding rules.

Supported Rules

As of R2023a, Polyspace supports these **Required** rules.

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A0-1-1	A project shall not contain instances of non-volatile variables being given values that are not subsequently used	`AUTOSAR C++14 Rule A0-1-1`
AUTOSAR C++14 Rule A0-1-2	The value returned by a function having a non-void return type that is not an overloaded operator shall be used	`AUTOSAR C++14 Rule A0-1-2`
AUTOSAR C++14 Rule A0-1-3	Every function defined in an anonymous namespace, or static function with internal linkage, or private member function shall be used	`AUTOSAR C++14 Rule A0-1-3`
AUTOSAR C++14 Rule A0-1-4	There shall be no unused named parameters in non-virtual functions	`AUTOSAR C++14 Rule A0-1-4`
AUTOSAR C++14 Rule A0-1-5	There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it	`AUTOSAR C++14 Rule A0-1-5`
AUTOSAR C++14 Rule A0-4-2	Type long double shall not be used	`AUTOSAR C++14 Rule A0-4-2`
AUTOSAR C++14 Rule A0-4-4	Range, domain and pole errors shall be checked when using math functions	`AUTOSAR C++14 Rule A0-4-4`
AUTOSAR C++14 Rule A1-1-1	All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features	`AUTOSAR C++14 Rule A1-1-1`
AUTOSAR C++14 Rule A10-1-1	Class shall not be derived from more than one base class which is not an interface class	`AUTOSAR C++14 Rule A10-1-1`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A10-2-1	Non-virtual public or protected member functions shall not be redefined in derived classes	`AUTOSAR C++14 Rule A10-2-1`
AUTOSAR C++14 Rule A10-3-1	Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final	`AUTOSAR C++14 Rule A10-3-1`
AUTOSAR C++14 Rule A10-3-2	Each overriding virtual function shall be declared with the override or final specifier	`AUTOSAR C++14 Rule A10-3-2`
AUTOSAR C++14 Rule A10-3-3	Virtual functions shall not be introduced in a final class	`AUTOSAR C++14 Rule A10-3-3`
AUTOSAR C++14 Rule A10-3-5	A user-defined assignment operator shall not be virtual	`AUTOSAR C++14 Rule A10-3-5`
AUTOSAR C++14 Rule A11-0-2	A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class	`AUTOSAR C++14 Rule A11-0-2`
AUTOSAR C++14 Rule A11-3-1	Friend declarations shall not be used	`AUTOSAR C++14 Rule A11-3-1`
AUTOSAR C++14 Rule A12-0-1	If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well	`AUTOSAR C++14 Rule A12-0-1`
AUTOSAR C++14 Rule A12-0-2	Bitwise operations and operations that assume data representation in memory shall not be performed on objects	`AUTOSAR C++14 Rule A12-0-2`
AUTOSAR C++14 Rule A12-1-1	Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members	`AUTOSAR C++14 Rule A12-1-1`
AUTOSAR C++14 Rule A12-1-2	Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type	`AUTOSAR C++14 Rule A12-1-2`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A12-1-3	If all user-defined constructors of a class initialize data members with constant values that are the same across all constructors, then data members shall be initialized using NSDMI instead	AUTOSAR C++14 Rule A12-1-3
AUTOSAR C++14 Rule A12-1-4	All constructors that are callable with a single argument of fundamental type shall be declared explicit	AUTOSAR C++14 Rule A12-1-4
AUTOSAR C++14 Rule A12-1-5	Common class initialization for non-constant members shall be done by a delegating constructor	AUTOSAR C++14 Rule A12-1-5
AUTOSAR C++14 Rule A12-1-6	Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors	AUTOSAR C++14 Rule A12-1-6
AUTOSAR C++14 Rule A12-4-1	Destructor of a base class shall be public virtual, public override or protected non-virtual	AUTOSAR C++14 Rule A12-4-1
AUTOSAR C++14 Rule A12-6-1	All class data members that are initialized by the constructor shall be initialized using member initializers	AUTOSAR C++14 Rule A12-6-1
AUTOSAR C++14 Rule A12-7-1	If the behavior of a user-defined special member function is identical to implicitly defined special member function, then it shall be defined "=default" or be left undefined	AUTOSAR C++14 Rule A12-7-1
AUTOSAR C++14 Rule A12-8-1	Move and copy constructors shall move and respectively copy base classes and data members of a class, without any side effects	AUTOSAR C++14 Rule A12-8-1
AUTOSAR C++14 Rule A12-8-3	Moved-from object shall not be read-accessed	AUTOSAR C++14 Rule A12-8-3
AUTOSAR C++14 Rule A12-8-4	Move constructor shall not initialize its class members and base classes using copy semantics	AUTOSAR C++14 Rule A12-8-4

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A12-8-5	A copy assignment and a move assignment operators shall handle self-assignment	`AUTOSAR C++14 Rule A12-8-5`
AUTOSAR C++14 Rule A12-8-6	Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class	`AUTOSAR C++14 Rule A12-8-6`
AUTOSAR C++14 Rule A13-1-2	User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters	`AUTOSAR C++14 Rule A13-1-2`
AUTOSAR C++14 Rule A13-1-3	User defined literals operators shall only perform conversion of passed parameters	`AUTOSAR C++14 Rule A13-1-3`
AUTOSAR C++14 Rule A13-2-1	An assignment operator shall return a reference to "this"	`AUTOSAR C++14 Rule A13-2-1`
AUTOSAR C++14 Rule A13-2-2	A binary arithmetic operator and a bitwise operator shall return a "prvalue"	`AUTOSAR C++14 Rule A13-2-2`
AUTOSAR C++14 Rule A13-2-3	A relational operator shall return a boolean value	`AUTOSAR C++14 Rule A13-2-3`
AUTOSAR C++14 Rule A13-3-1	A function that contains "forwarding reference" as its argument shall not be overloaded	`AUTOSAR C++14 Rule A13-3-1`
AUTOSAR C++14 Rule A13-5-1	If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented	`AUTOSAR C++14 Rule A13-5-1`
AUTOSAR C++14 Rule A13-5-2	All user-defined conversion operators shall be defined explicit	`AUTOSAR C++14 Rule A13-5-2`
AUTOSAR C++14 Rule A13-5-4	If two opposite operators are defined, one shall be defined in terms of the other	`AUTOSAR C++14 Rule A13-5-4`
AUTOSAR C++14 Rule A13-5-5	Comparison operators shall be non-member functions with identical parameter types and noexcept	`AUTOSAR C++14 Rule A13-5-5`
AUTOSAR C++14 Rule A13-6-1	Digit sequences separators ' shall only be used as follows: (1) for decimal, every 3 digits, (2) for hexadecimal, every 2 digits, (3) for binary, every 4 digits	`AUTOSAR C++14 Rule A13-6-1`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A14-5-1	A template constructor shall not participate in overload resolution for a single argument of the enclosing class type	`AUTOSAR C++14 Rule A14-5-1`
AUTOSAR C++14 Rule A14-7-1	A type used as a template argument shall provide all members that are used by the template	`AUTOSAR C++14 Rule A14-7-1`
AUTOSAR C++14 Rule A14-7-2	Template specialization shall be declared in the same file (1) as the primary template (2) as a user-defined type, for which the specialization is declared	`AUTOSAR C++14 Rule A14-7-2`
AUTOSAR C++14 Rule A14-8-2	Explicit specializations of function templates shall not be used	`AUTOSAR C++14 Rule A14-8-2`
AUTOSAR C++14 Rule A15-0-2	At least the basic guarantee for exception safety shall be provided for all operations. In addition, each function may offer either the strong guarantee or the nothrow guarantee	`AUTOSAR C++14 Rule A15-0-2`
AUTOSAR C++14 Rule A15-0-3	Exception safety guarantee of a called function shall be considered	`AUTOSAR C++14 Rule A15-0-3`
AUTOSAR C++14 Rule A15-0-7	Exception handling mechanism shall guarantee a deterministic worst-case time execution time	`AUTOSAR C++14 Rule A15-0-7`
AUTOSAR C++14 Rule A15-1-2	An exception object shall not be a pointer	`AUTOSAR C++14 Rule A15-1-2`
AUTOSAR C++14 Rule A15-1-4	If a function exits with an exception, then before a throw, the function shall place all objects/resources that the function constructed in valid states or it shall delete them.	`AUTOSAR C++14 Rule A15-1-4`
AUTOSAR C++14 Rule A15-1-5	Exceptions shall not be thrown across execution boundaries	`AUTOSAR C++14 Rule A15-1-5`
AUTOSAR C++14 Rule A15-2-1	Constructors that are not noexcept shall not be invoked before program startup	`AUTOSAR C++14 Rule A15-2-1`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A15-2-2	If a constructor is not noexcept and the constructor cannot finish object initialization, then it shall deallocate the object's resources and it shall throw an exception	AUTOSAR C++14 Rule A15-2-2
AUTOSAR C++14 Rule A15-3-3	Main function and a task main function shall catch at least: base class exceptions from all third-party libraries used, std::exception and all otherwise unhandled exceptions	AUTOSAR C++14 Rule A15-3-3
AUTOSAR C++14 Rule A15-3-4	Catch-all (ellipsis and std::exception) handlers shall be used only in (a) main, (b) task main functions, (c) in functions that are supposed to isolate independent components and (d) when calling third-party code that uses exceptions not according to AUTOSAR C++14 guidelines	AUTOSAR C++14 Rule A15-3-4
AUTOSAR C++14 Rule A15-3-5	A class type exception shall be caught by reference or const reference	AUTOSAR C++14 Rule A15-3-5
AUTOSAR C++14 Rule A15-4-1	Dynamic exception-specification shall not be used	AUTOSAR C++14 Rule A15-4-1
AUTOSAR C++14 Rule A15-4-2	If a function is declared to be noexcept, noexcept(true) or noexcept(<true condition>), then it shall not exit with an exception	AUTOSAR C++14 Rule A15-4-2
AUTOSAR C++14 Rule A15-4-3	The noexcept specification of a function shall either be identical across all translation units, or identical or more restrictive between a virtual member function and an overrider	AUTOSAR C++14 Rule A15-4-3
AUTOSAR C++14 Rule A15-4-4	A declaration of non-throwing function shall contain noexcept specification	AUTOSAR C++14 Rule A15-4-4

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A15-4-5	Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overriders	AUTOSAR C++14 Rule A15-4-5
AUTOSAR C++14 Rule A15-5-1	All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A noexcept exception specification shall be added to these functions as appropriate	AUTOSAR C++14 Rule A15-5-1
AUTOSAR C++14 Rule A15-5-2	Program shall not be abruptly terminated. In particular, an implicit or explicit invocation of std::abort(), std::quick_exit(), std::_Exit(), std::terminate() shall not be done	AUTOSAR C++14 Rule A15-5-2
AUTOSAR C++14 Rule A15-5-3	The std::terminate() function shall not be called implicitly	AUTOSAR C++14 Rule A15-5-3
AUTOSAR C++14 Rule A16-0-1	The preprocessor shall only be used for unconditional and conditional file inclusion and include guards, and using specific directives	AUTOSAR C++14 Rule A16-0-1
AUTOSAR C++14 Rule A16-2-1	The ', ", /*, //, \ characters shall not occur in a header file name or in #include directive	AUTOSAR C++14 Rule A16-2-1
AUTOSAR C++14 Rule A16-2-2	There shall be no unused include directives	AUTOSAR C++14 Rule A16-2-2
AUTOSAR C++14 Rule A16-2-3	An include directive shall be added explicitly for every symbol used in a file	AUTOSAR C++14 Rule A16-2-3
AUTOSAR C++14 Rule A16-6-1	#error directive shall not be used	AUTOSAR C++14 Rule A16-6-1
AUTOSAR C++14 Rule A16-7-1	The #pragma directive shall not be used	AUTOSAR C++14 Rule A16-7-1
AUTOSAR C++14 Rule A17-0-1	Reserved identifiers, macros and functions in the C++ standard library shall not be defined, redefined or undefined	AUTOSAR C++14 Rule A17-0-1

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A17-1-1	Use of the C Standard Library shall be encapsulated and isolated	AUTOSAR C++14 Rule A17-1-1
AUTOSAR C++14 Rule A17-6-1	Non-standard entities shall not be added to standard namespaces	AUTOSAR C++14 Rule A17-6-1
AUTOSAR C++14 Rule A18-0-1	The C library facilities shall only be accessed through C++ library headers	AUTOSAR C++14 Rule A18-0-1
AUTOSAR C++14 Rule A18-0-2	The error state of a conversion from string to a numeric value shall be checked	AUTOSAR C++14 Rule A18-0-2
AUTOSAR C++14 Rule A18-0-3	The library <clocale> (locale.h) and the setlocale function shall not be used	AUTOSAR C++14 Rule A18-0-3
AUTOSAR C++14 Rule A18-1-1	C-style arrays shall not be used	AUTOSAR C++14 Rule A18-1-1
AUTOSAR C++14 Rule A18-1-2	The std::vector<bool> specialization shall not be used	AUTOSAR C++14 Rule A18-1-2
AUTOSAR C++14 Rule A18-1-3	The std::auto_ptr shall not be used	AUTOSAR C++14 Rule A18-1-3
AUTOSAR C++14 Rule A18-1-4	A pointer pointing to an element of an array of objects shall not be passed to a smart pointer of single object type	AUTOSAR C++14 Rule A18-1-4
AUTOSAR C++14 Rule A18-1-6	All std::hash specializations for user-defined types shall have a noexcept function call operator	AUTOSAR C++14 Rule A18-1-6
AUTOSAR C++14 Rule A18-5-1	Functions malloc, calloc, realloc and free shall not be used	AUTOSAR C++14 Rule A18-5-1
AUTOSAR C++14 Rule A18-5-10	Placement new shall be used only with properly aligned pointers to sufficient storage capacity	AUTOSAR C++14 Rule A18-5-10
AUTOSAR C++14 Rule A18-5-11	"operator new" and "operator delete" shall be defined together	AUTOSAR C++14 Rule A18-5-11
AUTOSAR C++14 Rule A18-5-2	Non-placement new or delete expressions shall not be used	AUTOSAR C++14 Rule A18-5-2
AUTOSAR C++14 Rule A18-5-3	The form of delete operator shall match the form of new operator used to allocate the memory	AUTOSAR C++14 Rule A18-5-3

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A18-5-4	If a project has sized or unsized version of operator 'delete' globally defined, then both sized and unsized versions shall be defined	AUTOSAR C++14 Rule A18-5-4
AUTOSAR C++14 Rule A18-5-5	Memory management functions shall ensure the following: (a) deterministic behavior resulting with the existence of worst-case execution time, (b) avoiding memory fragmentation, (c) avoid running out of memory, (d) avoiding mismatched allocations or deallocations, (e) no dependence on non-deterministic calls to kernel	AUTOSAR C++14 Rule A18-5-5
AUTOSAR C++14 Rule A18-5-7	If non-real-time implementation of dynamic memory management functions is used in the project, then memory shall only be allocated and deallocated during non-real-time program phases	AUTOSAR C++14 Rule A18-5-7
AUTOSAR C++14 Rule A18-5-8	Objects that do not outlive a function shall have automatic storage duration	AUTOSAR C++14 Rule A18-5-8
AUTOSAR C++14 Rule A18-5-9	Custom implementations of dynamic memory allocation and deallocation functions shall meet the semantic requirements specified in the corresponding "Required behaviour" clause from the C++ Standard	AUTOSAR C++14 Rule A18-5-9
AUTOSAR C++14 Rule A18-9-1	The std::bind shall not be used	AUTOSAR C++14 Rule A18-9-1
AUTOSAR C++14 Rule A18-9-2	Forwarding values to other functions shall be done via: (1) std::move if the value is an rvalue reference, (2) std::forward if the value is forwarding reference	AUTOSAR C++14 Rule A18-9-2
AUTOSAR C++14 Rule A18-9-3	The std::move shall not be used on objects declared const or const&	AUTOSAR C++14 Rule A18-9-3
AUTOSAR C++14 Rule A18-9-4	An argument to std::forward shall not be subsequently used	AUTOSAR C++14 Rule A18-9-4

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A2-10-1	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope	AUTOSAR C++14 Rule A2-10-1
AUTOSAR C++14 Rule A2-10-4	The identifier name of a non-member object with static storage duration or static function shall not be reused within a namespace	AUTOSAR C++14 Rule A2-10-4
AUTOSAR C++14 Rule A2-10-6	A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope	AUTOSAR C++14 Rule A2-10-6
AUTOSAR C++14 Rule A2-11-1	Volatile keyword shall not be used	AUTOSAR C++14 Rule A2-11-1
AUTOSAR C++14 Rule A2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2014 shall be used	AUTOSAR C++14 Rule A2-13-1
AUTOSAR C++14 Rule A2-13-2	String literals with different encoding prefixes shall not be concatenated	AUTOSAR C++14 Rule A2-13-2
AUTOSAR C++14 Rule A2-13-3	Type wchar_t shall not be used	AUTOSAR C++14 Rule A2-13-3
AUTOSAR C++14 Rule A2-13-4	String literals shall not be assigned to non-constant pointers	AUTOSAR C++14 Rule A2-13-4
AUTOSAR C++14 Rule A2-13-6	Universal character names shall be used only inside character or string literals	AUTOSAR C++14 Rule A2-13-6
AUTOSAR C++14 Rule A2-3-1	Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code	AUTOSAR C++14 Rule A2-3-1
AUTOSAR C++14 Rule A2-5-1	Trigraphs shall not be used	AUTOSAR C++14 Rule A2-5-1
AUTOSAR C++14 Rule A2-5-2	Digraphs shall not be used	AUTOSAR C++14 Rule A2-5-2
AUTOSAR C++14 Rule A2-7-1	The character \ shall not occur as a last character of a C++ comment	AUTOSAR C++14 Rule A2-7-1
AUTOSAR C++14 Rule A2-7-2	Sections of code shall not be "commented out"	AUTOSAR C++14 Rule A2-7-2

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A2-7-3	All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation	AUTOSAR C++14 Rule A2-7-3
AUTOSAR C++14 Rule A2-8-1	A header file name should reflect the logical entity for which it provides declarations.	AUTOSAR C++14 Rule A2-8-1
AUTOSAR C++14 Rule A20-8-1	An already-owned pointer value shall not be stored in an unrelated smart pointer	AUTOSAR C++14 Rule A20-8-1
AUTOSAR C++14 Rule A20-8-2	A std::unique_ptr shall be used to represent exclusive ownership	AUTOSAR C++14 Rule A20-8-2
AUTOSAR C++14 Rule A20-8-3	A std::shared_ptr shall be used to represent shared ownership	AUTOSAR C++14 Rule A20-8-3
AUTOSAR C++14 Rule A20-8-4	A std::unique_ptr shall be used over std::shared_ptr if ownership sharing is not required	AUTOSAR C++14 Rule A20-8-4
AUTOSAR C++14 Rule A20-8-5	std::make_unique shall be used to construct objects owned by std::unique_ptr	AUTOSAR C++14 Rule A20-8-5
AUTOSAR C++14 Rule A20-8-6	std::make_shared shall be used to construct objects owned by std::shared_ptr	AUTOSAR C++14 Rule A20-8-6
AUTOSAR C++14 Rule A20-8-7	A std::weak_ptr shall be used to represent temporary shared ownership.	AUTOSAR C++14 Rule A20-8-7
AUTOSAR C++14 Rule A21-8-1	Arguments to character-handling functions shall be representable as an unsigned char	AUTOSAR C++14 Rule A21-8-1
AUTOSAR C++14 Rule A23-0-1	An iterator shall not be implicitly converted to const_iterator	AUTOSAR C++14 Rule A23-0-1
AUTOSAR C++14 Rule A23-0-2	Elements of a container shall only be accessed via valid references, iterators, and pointers	AUTOSAR C++14 Rule A23-0-2
AUTOSAR C++14 Rule A25-1-1	Non-static data members or captured values of predicate function objects that are state related to this object's identity shall not be copied	AUTOSAR C++14 Rule A25-1-1

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A25-4-1	Ordering predicates used with associative containers and STL sorting and related algorithms shall adhere to a strict weak ordering relation	`AUTOSAR C++14 Rule A25-4-1`
AUTOSAR C++14 Rule A26-5-1	Pseudorandom numbers shall not be generated using std::rand()	`AUTOSAR C++14 Rule A26-5-1`
AUTOSAR C++14 Rule A26-5-2	Random number engines shall not be default-initialized	`AUTOSAR C++14 Rule A26-5-2`
AUTOSAR C++14 Rule A27-0-1	Inputs from independent components shall be validated.	`AUTOSAR C++14 Rule A27-0-1`
AUTOSAR C++14 Rule A27-0-3	Alternate input and output operations on a file stream shall not be used without an intervening flush or positioning call	`AUTOSAR C++14 Rule A27-0-3`
AUTOSAR C++14 Rule A27-0-4	C-style strings shall not be used	`AUTOSAR C++14 Rule A27-0-4`
AUTOSAR C++14 Rule A3-1-1	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule	`AUTOSAR C++14 Rule A3-1-1`
AUTOSAR C++14 Rule A3-1-2	Header files, that are defined locally in the project, shall have a file name extension of one of: `.h`, `.hpp` or `.hxx`	`AUTOSAR C++14 Rule A3-1-2`
AUTOSAR C++14 Rule A3-1-4	When an array with external linkage is declared, its size shall be stated explicitly	`AUTOSAR C++14 Rule A3-1-4`
AUTOSAR C++14 Rule A3-1-5	A function definition shall only be placed in a class definition if (1) the function is intended to be inlined (2) it is a member function template (3) it is a member function of a class template	`AUTOSAR C++14 Rule A3-1-5`
AUTOSAR C++14 Rule A3-3-1	Objects or functions with external linkage (including members of named namespaces) shall be declared in a header file	`AUTOSAR C++14 Rule A3-3-1`
AUTOSAR C++14 Rule A3-3-2	Static and thread-local objects shall be constant-initialized	`AUTOSAR C++14 Rule A3-3-2`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A3-8-1	An object shall not be accessed outside of its lifetime	AUTOSAR C++14 Rule A3-8-1
AUTOSAR C++14 Rule A3-9-1	Fixed width integer types from <cstdint>, indicating the size and signedness, shall be used in place of the basic numerical types	AUTOSAR C++14 Rule A3-9-1
AUTOSAR C++14 Rule A4-10-1	Only nullptr literal shall be used as the null-pointer-constraint	AUTOSAR C++14 Rule A4-10-1
AUTOSAR C++14 Rule A4-5-1	Expressions with type enum or enum class shall not be used as operands to built-in and overloaded operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=	AUTOSAR C++14 Rule A4-5-1
AUTOSAR C++14 Rule A4-7-1	An integer expression shall not lead to data loss	AUTOSAR C++14 Rule A4-7-1
AUTOSAR C++14 Rule A5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits	AUTOSAR C++14 Rule A5-0-1
AUTOSAR C++14 Rule A5-0-2	The condition of an if-statement and the condition of an iteration statement shall have type bool	AUTOSAR C++14 Rule A5-0-2
AUTOSAR C++14 Rule A5-0-3	The declaration of objects shall contain no more than two levels of pointer indirection	AUTOSAR C++14 Rule A5-0-3
AUTOSAR C++14 Rule A5-0-4	Pointer arithmetic shall not be used with pointers to non-final classes	AUTOSAR C++14 Rule A5-0-4
AUTOSAR C++14 Rule A5-1-1	Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead	AUTOSAR C++14 Rule A5-1-1
AUTOSAR C++14 Rule A5-1-2	Variables shall not be implicitly captured in a lambda expression	AUTOSAR C++14 Rule A5-1-2
AUTOSAR C++14 Rule A5-1-3	Parameter list (possibly empty) shall be included in every lambda expression	AUTOSAR C++14 Rule A5-1-3

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A5-1-4	A lambda expression object shall not outlive any of its reference-captured objects	AUTOSAR C++14 Rule A5-1-4
AUTOSAR C++14 Rule A5-1-7	A lambda shall not be an operand to decltype or typeid	AUTOSAR C++14 Rule A5-1-7
AUTOSAR C++14 Rule A5-10-1	A pointer to member virtual function shall only be tested for equality with null-pointer-constant	AUTOSAR C++14 Rule A5-10-1
AUTOSAR C++14 Rule A5-16-1	The ternary conditional operator shall not be used as a sub-expression	AUTOSAR C++14 Rule A5-16-1
AUTOSAR C++14 Rule A5-2-2	Traditional C-style casts shall not be used	AUTOSAR C++14 Rule A5-2-2
AUTOSAR C++14 Rule A5-2-3	A cast shall not remove any const or volatile qualification from the type of a pointer or reference	AUTOSAR C++14 Rule A5-2-3
AUTOSAR C++14 Rule A5-2-4	reinterpret_cast shall not be used	AUTOSAR C++14 Rule A5-2-4
AUTOSAR C++14 Rule A5-2-5	An array or container shall not be accessed beyond its range	AUTOSAR C++14 Rule A5-2-5
AUTOSAR C++14 Rule A5-2-6	The operands of a logical && or || shall be parenthesized if the operands contain binary operators	AUTOSAR C++14 Rule A5-2-6
AUTOSAR C++14 Rule A5-3-1	Evaluation of the operand to the typeid operator shall not contain side effects	AUTOSAR C++14 Rule A5-3-1
AUTOSAR C++14 Rule A5-3-2	Null pointers shall not be dereferenced	AUTOSAR C++14 Rule A5-3-2
AUTOSAR C++14 Rule A5-3-3	Pointers to incomplete class types shall not be deleted	AUTOSAR C++14 Rule A5-3-3
AUTOSAR C++14 Rule A5-5-1	A pointer to member shall not access non-existent class members	AUTOSAR C++14 Rule A5-5-1
AUTOSAR C++14 Rule A5-6-1	The right hand operand of the integer division or remainder operators shall not be equal to zero	AUTOSAR C++14 Rule A5-6-1
AUTOSAR C++14 Rule A6-2-1	Move and copy assignment operators shall either move or respectively copy base classes and data members of a class, without any side effects	AUTOSAR C++14 Rule A6-2-1

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A6-2-2	Expression statements shall not be explicit calls to constructors of temporary objects only	`AUTOSAR C++14 Rule A6-2-2`
AUTOSAR C++14 Rule A6-4-1	A `switch` statement shall have at least two case-clauses, distinct from the default label	`AUTOSAR C++14 Rule A6-4-1`
AUTOSAR C++14 Rule A6-5-1	A for-loop that loops through all elements of the container and does not use its loop-counter shall not be used	`AUTOSAR C++14 Rule A6-5-1`
AUTOSAR C++14 Rule A6-5-2	A `for` loop shall contain a single loop-counter which shall not have floating-point type	`AUTOSAR C++14 Rule A6-5-2`
AUTOSAR C++14 Rule A6-6-1	The `goto` statement shall not be used	`AUTOSAR C++14 Rule A6-6-1`
AUTOSAR C++14 Rule A7-1-1	Constexpr or const specifiers shall be used for immutable data declaration	`AUTOSAR C++14 Rule A7-1-1`
AUTOSAR C++14 Rule A7-1-2	The constexpr specifier shall be used for values that can be determined at compile time	`AUTOSAR C++14 Rule A7-1-2`
AUTOSAR C++14 Rule A7-1-3	CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name	`AUTOSAR C++14 Rule A7-1-3`
AUTOSAR C++14 Rule A7-1-4	The register keyword shall not be used	`AUTOSAR C++14 Rule A7-1-4`
AUTOSAR C++14 Rule A7-1-5	The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax	`AUTOSAR C++14 Rule A7-1-5`
AUTOSAR C++14 Rule A7-1-6	The typedef specifier shall not be used	`AUTOSAR C++14 Rule A7-1-6`
AUTOSAR C++14 Rule A7-1-7	Each expression statement and identifier declaration shall be placed on a separate line	`AUTOSAR C++14 Rule A7-1-7`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A7-1-8	A non-type specifier shall be placed before a type specifier in a declaration	AUTOSAR C++14 Rule A7-1-8
AUTOSAR C++14 Rule A7-1-9	A class, structure, or enumeration shall not be declared in the definition of its type	AUTOSAR C++14 Rule A7-1-9
AUTOSAR C++14 Rule A7-2-1	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration	AUTOSAR C++14 Rule A7-2-1
AUTOSAR C++14 Rule A7-2-2	Enumeration underlying type shall be explicitly defined	AUTOSAR C++14 Rule A7-2-2
AUTOSAR C++14 Rule A7-2-3	Enumerations shall be declared as scoped enum classes	AUTOSAR C++14 Rule A7-2-3
AUTOSAR C++14 Rule A7-2-4	In an enumeration, either (1) none, (2) the first or (3) all enumerators shall be initialized	AUTOSAR C++14 Rule A7-2-4
AUTOSAR C++14 Rule A7-3-1	All overloads of a function shall be visible from where it is called	AUTOSAR C++14 Rule A7-3-1
AUTOSAR C++14 Rule A7-4-1	The asm declaration shall not be used	AUTOSAR C++14 Rule A7-4-1
AUTOSAR C++14 Rule A7-5-1	A function shall not return a reference or a pointer to a parameter that is passed by reference to const	AUTOSAR C++14 Rule A7-5-1
AUTOSAR C++14 Rule A7-5-2	Functions shall not call themselves, either directly or indirectly	AUTOSAR C++14 Rule A7-5-2
AUTOSAR C++14 Rule A7-6-1	Functions declared with the [[noreturn]] attribute shall not return	AUTOSAR C++14 Rule A7-6-1
AUTOSAR C++14 Rule A8-2-1	When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters	AUTOSAR C++14 Rule A8-2-1
AUTOSAR C++14 Rule A8-4-1	Functions shall not be defined using the ellipsis notation	AUTOSAR C++14 Rule A8-4-1
AUTOSAR C++14 Rule A8-4-10	A parameter shall be passed by reference if it can't be NULL	AUTOSAR C++14 Rule A8-4-10
AUTOSAR C++14 Rule A8-4-11	A smart pointer shall only be used as a parameter type if it expresses lifetime semantics	AUTOSAR C++14 Rule A8-4-11

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A8-4-12	A std::unique_ptr shall be passed to a function as: (1) a copy to express the function assumes ownership (2) an lvalue reference to express that the function replaces the managed object.	AUTOSAR C++14 Rule A8-4-12
AUTOSAR C++14 Rule A8-4-13	A std::shared_ptr shall be passed to a function as: (1) a copy to express the function shares ownership (2) an lvalue reference to express that the function replaces the managed object (3) a `const` lvalue reference to express that the function retains a reference count.	AUTOSAR C++14 Rule A8-4-13
AUTOSAR C++14 Rule A8-4-14	Interfaces shall be precisely and strongly typed	AUTOSAR C++14 Rule A8-4-14
AUTOSAR C++14 Rule A8-4-2	All exit paths from a function with non-void return type shall have an explicit return statement with an expression	AUTOSAR C++14 Rule A8-4-2
AUTOSAR C++14 Rule A8-4-5	"consume" parameters declared as X && shall always be moved from	AUTOSAR C++14 Rule A8-4-5
AUTOSAR C++14 Rule A8-4-6	"forward" parameters declared as T && shall always be forwarded	AUTOSAR C++14 Rule A8-4-6
AUTOSAR C++14 Rule A8-4-7	"in" parameters for "cheap to copy" types shall be passed by value	AUTOSAR C++14 Rule A8-4-7
AUTOSAR C++14 Rule A8-4-8	Output parameters shall not be used	AUTOSAR C++14 Rule A8-4-8
AUTOSAR C++14 Rule A8-4-9	"in-out" parameters declared as T & shall be modified	AUTOSAR C++14 Rule A8-4-9
AUTOSAR C++14 Rule A8-5-0	All memory shall be initialized before it is read	AUTOSAR C++14 Rule A8-5-0

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A8-5-1	In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition	`AUTOSAR C++14 Rule A8-5-1`
AUTOSAR C++14 Rule A8-5-2	Braced-initialization {}, without equals sign, shall be used for variable initialization	`AUTOSAR C++14 Rule A8-5-2`
AUTOSAR C++14 Rule A8-5-3	A variable of type auto shall not be initialized using {} or ={} braced-initialization	`AUTOSAR C++14 Rule A8-5-3`
AUTOSAR C++14 Rule A9-3-1	Member functions shall not return non-constant "raw" pointers or references to private or protected data owned by the class	`AUTOSAR C++14 Rule A9-3-1`
AUTOSAR C++14 Rule A9-5-1	Unions shall not be used	`AUTOSAR C++14 Rule A9-5-1`
AUTOSAR C++14 Rule A9-6-1	Data types used for interfacing with hardware or conforming to communication protocols shall be trivial, standard-layout and only contain members of types with defined sizes	`AUTOSAR C++14 Rule A9-6-1`
AUTOSAR C++14 Rule M0-1-1	A project shall not contain unreachable code	`AUTOSAR C++14 Rule M0-1-1`
AUTOSAR C++14 Rule M0-1-2	A project shall not contain infeasible paths	`AUTOSAR C++14 Rule M0-1-2`
AUTOSAR C++14 Rule M0-1-3	A project shall not contain unused variables	`AUTOSAR C++14 Rule M0-1-3`
AUTOSAR C++14 Rule M0-1-4	A project shall not contain non-volatile POD variables having only one use	`AUTOSAR C++14 Rule M0-1-4`
AUTOSAR C++14 Rule M0-1-8	All functions with void return type shall have external side effect(s)	`AUTOSAR C++14 Rule M0-1-8`
AUTOSAR C++14 Rule M0-1-9	There shall be no dead code	`AUTOSAR C++14 Rule M0-1-9`
AUTOSAR C++14 Rule M0-2-1	An object shall not be assigned to an overlapping object	`AUTOSAR C++14 Rule M0-2-1`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M0-3-2	If a function generates error information, then that error information shall be tested	AUTOSAR C++14 Rule M0-3-2
AUTOSAR C++14 Rule M10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy	AUTOSAR C++14 Rule M10-1-2
AUTOSAR C++14 Rule M10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy	AUTOSAR C++14 Rule M10-1-3
AUTOSAR C++14 Rule M10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual	AUTOSAR C++14 Rule M10-3-3
AUTOSAR C++14 Rule M11-0-1	Member data in non-POD class types shall be private	AUTOSAR C++14 Rule M11-0-1
AUTOSAR C++14 Rule M12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor	AUTOSAR C++14 Rule M12-1-1
AUTOSAR C++14 Rule M14-5-3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter	AUTOSAR C++14 Rule M14-5-3
AUTOSAR C++14 Rule M14-6-1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->	AUTOSAR C++14 Rule M14-6-1
AUTOSAR C++14 Rule M15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement	AUTOSAR C++14 Rule M15-0-3
AUTOSAR C++14 Rule M15-1-1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown	AUTOSAR C++14 Rule M15-1-1
AUTOSAR C++14 Rule M15-1-2	NULL shall not be thrown explicitly	AUTOSAR C++14 Rule M15-1-2
AUTOSAR C++14 Rule M15-1-3	An empty throw (throw;) shall only be used in the compound statement of a catch handler	AUTOSAR C++14 Rule M15-1-3
AUTOSAR C++14 Rule M15-3-1	Exceptions shall be raised only after start-up and before termination	AUTOSAR C++14 Rule M15-3-1

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases	`AUTOSAR C++14 Rule M15-3-3`
AUTOSAR C++14 Rule M15-3-4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point	`AUTOSAR C++14 Rule M15-3-4`
AUTOSAR C++14 Rule M15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class	`AUTOSAR C++14 Rule M15-3-6`
AUTOSAR C++14 Rule M15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last	`AUTOSAR C++14 Rule M15-3-7`
AUTOSAR C++14 Rule M16-0-1	#include directives in a file shall only be preceded by other preprocessor directives or comments	`AUTOSAR C++14 Rule M16-0-1`
AUTOSAR C++14 Rule M16-0-2	Macros shall only be #define'd or #undef'd in the global namespace	`AUTOSAR C++14 Rule M16-0-2`
AUTOSAR C++14 Rule M16-0-5	Arguments to a function-like macro shall not contain tokens that look like pre-processing directives	`AUTOSAR C++14 Rule M16-0-5`
AUTOSAR C++14 Rule M16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##	`AUTOSAR C++14 Rule M16-0-6`
AUTOSAR C++14 Rule M16-0-7	Undefined macro identifiers shall not be used in #if or #elif pre-processor directives, except as operands to the defined operator	`AUTOSAR C++14 Rule M16-0-7`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M16-0-8	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token	AUTOSAR C++14 Rule M16-0-8
AUTOSAR C++14 Rule M16-1-1	The defined pre-processor operator shall only be used in one of the two standard forms	AUTOSAR C++14 Rule M16-1-1
AUTOSAR C++14 Rule M16-1-2	All #else, #elif and #endif pre-processor directives shall reside in the same file as the #if or #ifdef directive to which they are related	AUTOSAR C++14 Rule M16-1-2
AUTOSAR C++14 Rule M16-2-3	Include guards shall be provided	AUTOSAR C++14 Rule M16-2-3
AUTOSAR C++14 Rule M16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition	AUTOSAR C++14 Rule M16-3-1
AUTOSAR C++14 Rule M17-0-2	The names of standard library macros and objects shall not be reused	AUTOSAR C++14 Rule M17-0-2
AUTOSAR C++14 Rule M17-0-3	The names of standard library functions shall not be overridden	AUTOSAR C++14 Rule M17-0-3
AUTOSAR C++14 Rule M17-0-5	The setjmp macro and the longjmp function shall not be used	AUTOSAR C++14 Rule M17-0-5
AUTOSAR C++14 Rule M18-0-3	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used	AUTOSAR C++14 Rule M18-0-3
AUTOSAR C++14 Rule M18-0-4	The time handling functions of library <ctime> shall not be used	AUTOSAR C++14 Rule M18-0-4
AUTOSAR C++14 Rule M18-0-5	The unbounded functions of library <cstring> shall not be used	AUTOSAR C++14 Rule M18-0-5
AUTOSAR C++14 Rule M18-2-1	The macro offsetof shall not be used	AUTOSAR C++14 Rule M18-2-1
AUTOSAR C++14 Rule M18-7-1	The signal handling facilities of <csignal> shall not be used	AUTOSAR C++14 Rule M18-7-1
AUTOSAR C++14 Rule M19-3-1	The error indicator errno shall not be used	AUTOSAR C++14 Rule M19-3-1
AUTOSAR C++14 Rule M2-10-1	Different identifiers shall be typographically unambiguous	AUTOSAR C++14 Rule M2-10-1

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M2-13-2	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used	AUTOSAR C++14 Rule M2-13-2
AUTOSAR C++14 Rule M2-13-3	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type	AUTOSAR C++14 Rule M2-13-3
AUTOSAR C++14 Rule M2-13-4	Literal suffixes shall be upper case	AUTOSAR C++14 Rule M2-13-4
AUTOSAR C++14 Rule M2-7-1	The character sequence /* shall not be used within a C-style comment	AUTOSAR C++14 Rule M2-7-1
AUTOSAR C++14 Rule M27-0-1	The stream input/output library <cstdio> shall not be used	AUTOSAR C++14 Rule M27-0-1
AUTOSAR C++14 Rule M3-1-2	Functions shall not be declared at block scope	AUTOSAR C++14 Rule M3-1-2
AUTOSAR C++14 Rule M3-2-1	All declarations of an object or function shall have compatible types	AUTOSAR C++14 Rule M3-2-1
AUTOSAR C++14 Rule M3-2-2	The One Definition Rule shall not be violated	AUTOSAR C++14 Rule M3-2-2
AUTOSAR C++14 Rule M3-2-3	A type, object or function that is used in multiple translation units shall be declared in one and only one file	AUTOSAR C++14 Rule M3-2-3
AUTOSAR C++14 Rule M3-2-4	An identifier with external linkage shall have exactly one definition	AUTOSAR C++14 Rule M3-2-4
AUTOSAR C++14 Rule M3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier	AUTOSAR C++14 Rule M3-3-2
AUTOSAR C++14 Rule M3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility	AUTOSAR C++14 Rule M3-4-1
AUTOSAR C++14 Rule M3-9-1	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations	AUTOSAR C++14 Rule M3-9-1
AUTOSAR C++14 Rule M3-9-3	The underlying bit representations of floating-point values shall not be used	AUTOSAR C++14 Rule M3-9-3

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M4-10-1	NULL shall not be used as an integer value	AUTOSAR C++14 Rule M4-10-1
AUTOSAR C++14 Rule M4-10-2	Literal zero (0) shall not be used as the null-pointer-constant	AUTOSAR C++14 Rule M4-10-2
AUTOSAR C++14 Rule M4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and ! =, the unary & operator, and the conditional operator	AUTOSAR C++14 Rule M4-5-1
AUTOSAR C++14 Rule M4-5-3	Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and ! =, and the unary & operator	AUTOSAR C++14 Rule M4-5-3
AUTOSAR C++14 Rule M5-0-10	If the bitwise operators ~and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand	AUTOSAR C++14 Rule M5-0-10
AUTOSAR C++14 Rule M5-0-11	The plain char type shall only be used for the storage and use of character values	AUTOSAR C++14 Rule M5-0-11
AUTOSAR C++14 Rule M5-0-12	Signed char and unsigned char type shall only be used for the storage and use of numeric values	AUTOSAR C++14 Rule M5-0-12
AUTOSAR C++14 Rule M5-0-14	The first operand of a conditional-operator shall have type bool	AUTOSAR C++14 Rule M5-0-14
AUTOSAR C++14 Rule M5-0-15	Array indexing shall be the only form of pointer arithmetic	AUTOSAR C++14 Rule M5-0-15
AUTOSAR C++14 Rule M5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array	AUTOSAR C++14 Rule M5-0-16

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M5-0-17	Subtraction between pointers shall only be applied to pointers that address elements of the same array	`AUTOSAR C++14 Rule M5-0-17`
AUTOSAR C++14 Rule M5-0-18	`>`, `>=`, `<`, `<=` shall not be applied to objects of pointer type, except where they point to the same array	`AUTOSAR C++14 Rule M5-0-18`
AUTOSAR C++14 Rule M5-0-20	Non-constant operands to a binary bitwise operator shall have the same underlying type	`AUTOSAR C++14 Rule M5-0-20`
AUTOSAR C++14 Rule M5-0-21	Bitwise operators shall only be applied to operands of unsigned underlying type	`AUTOSAR C++14 Rule M5-0-21`
AUTOSAR C++14 Rule M5-0-3	A cvalue expression shall not be implicitly converted to a different underlying type	`AUTOSAR C++14 Rule M5-0-3`
AUTOSAR C++14 Rule M5-0-4	An implicit integral conversion shall not change the signedness of the underlying type	`AUTOSAR C++14 Rule M5-0-4`
AUTOSAR C++14 Rule M5-0-5	There shall be no implicit floating-integral conversions	`AUTOSAR C++14 Rule M5-0-5`
AUTOSAR C++14 Rule M5-0-6	An implicit integral or floating-point conversion shall not reduce the size of the underlying type	`AUTOSAR C++14 Rule M5-0-6`
AUTOSAR C++14 Rule M5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression	`AUTOSAR C++14 Rule M5-0-7`
AUTOSAR C++14 Rule M5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression	`AUTOSAR C++14 Rule M5-0-8`
AUTOSAR C++14 Rule M5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression	`AUTOSAR C++14 Rule M5-0-9`
AUTOSAR C++14 Rule M5-14-1	The right hand operand of a logical &&, || operators shall not contain side effects	`AUTOSAR C++14 Rule M5-14-1`
AUTOSAR C++14 Rule M5-18-1	The comma operator shall not be used	`AUTOSAR C++14 Rule M5-18-1`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M5-19-1	Evaluation of constant unsigned integer expressions shall not lead to wrap-around	AUTOSAR C++14 Rule M5-19-1
AUTOSAR C++14 Rule M5-2-10	The increment (++) and decrement (--) operators shall not be mixed with other operators in an expression	AUTOSAR C++14 Rule M5-2-10
AUTOSAR C++14 Rule M5-2-11	The comma operator, && operator and the || operator shall not be overloaded	AUTOSAR C++14 Rule M5-2-11
AUTOSAR C++14 Rule M5-2-12	An identifier with array type passed as a function argument shall not decay to a pointer	AUTOSAR C++14 Rule M5-2-12
AUTOSAR C++14 Rule M5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of `dynamic_cast`	AUTOSAR C++14 Rule M5-2-2
AUTOSAR C++14 Rule M5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type	AUTOSAR C++14 Rule M5-2-6
AUTOSAR C++14 Rule M5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type	AUTOSAR C++14 Rule M5-2-8
AUTOSAR C++14 Rule M5-2-9	A cast shall not convert a pointer type to an integral type	AUTOSAR C++14 Rule M5-2-9
AUTOSAR C++14 Rule M5-3-1	Each operand of the ! operator, the logical && or the logical || operators shall have type bool	AUTOSAR C++14 Rule M5-3-1
AUTOSAR C++14 Rule M5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned	AUTOSAR C++14 Rule M5-3-2
AUTOSAR C++14 Rule M5-3-3	The unary & operator shall not be overloaded	AUTOSAR C++14 Rule M5-3-3
AUTOSAR C++14 Rule M5-3-4	Evaluation of the operand to the sizeof operator shall not contain side effects	AUTOSAR C++14 Rule M5-3-4
AUTOSAR C++14 Rule M5-8-1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand	AUTOSAR C++14 Rule M5-8-1

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M6-2-1	Assignment operators shall not be used in sub-expressions	AUTOSAR C++14 Rule M6-2-1
AUTOSAR C++14 Rule M6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality	AUTOSAR C++14 Rule M6-2-2
AUTOSAR C++14 Rule M6-2-3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character	AUTOSAR C++14 Rule M6-2-3
AUTOSAR C++14 Rule M6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement	AUTOSAR C++14 Rule M6-3-1
AUTOSAR C++14 Rule M6-4-1	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement	AUTOSAR C++14 Rule M6-4-1
AUTOSAR C++14 Rule M6-4-2	All if ... else if constructs shall be terminated with an else clause	AUTOSAR C++14 Rule M6-4-2
AUTOSAR C++14 Rule M6-4-3	A switch statement shall be a well-formed switch statement	AUTOSAR C++14 Rule M6-4-3
AUTOSAR C++14 Rule M6-4-4	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	AUTOSAR C++14 Rule M6-4-4
AUTOSAR C++14 Rule M6-4-5	An unconditional throw or break statement shall terminate every non-empty switch-clause	AUTOSAR C++14 Rule M6-4-5
AUTOSAR C++14 Rule M6-4-6	The final clause of a `switch` statement shall be the default-clause	AUTOSAR C++14 Rule M6-4-6
AUTOSAR C++14 Rule M6-4-7	The condition of a switch statement shall not have bool type	AUTOSAR C++14 Rule M6-4-7
AUTOSAR C++14 Rule M6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=	AUTOSAR C++14 Rule M6-5-2

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M6-5-3	The loop-counter shall not be modified within condition or statement	`AUTOSAR C++14 Rule M6-5-3`
AUTOSAR C++14 Rule M6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop	`AUTOSAR C++14 Rule M6-5-4`
AUTOSAR C++14 Rule M6-5-5	A loop-control-variable other than the loop-counter shall not be modified within condition or expression	`AUTOSAR C++14 Rule M6-5-5`
AUTOSAR C++14 Rule M6-5-6	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool	`AUTOSAR C++14 Rule M6-5-6`
AUTOSAR C++14 Rule M6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement	`AUTOSAR C++14 Rule M6-6-1`
AUTOSAR C++14 Rule M6-6-2	The goto statement shall jump to a label declared later in the same function body	`AUTOSAR C++14 Rule M6-6-2`
AUTOSAR C++14 Rule M6-6-3	The `continue` statement shall only be used within a well-formed `for` loop	`AUTOSAR C++14 Rule M6-6-3`
AUTOSAR C++14 Rule M7-1-2	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified	`AUTOSAR C++14 Rule M7-1-2`
AUTOSAR C++14 Rule M7-3-1	The global namespace shall only contain main, namespace declarations and extern "C" declarations	`AUTOSAR C++14 Rule M7-3-1`
AUTOSAR C++14 Rule M7-3-2	The identifier main shall not be used for a function other than the global function main	`AUTOSAR C++14 Rule M7-3-2`
AUTOSAR C++14 Rule M7-3-3	There shall be no unnamed namespaces in header files	`AUTOSAR C++14 Rule M7-3-3`
AUTOSAR C++14 Rule M7-3-4	Using-directives shall not be used	`AUTOSAR C++14 Rule M7-3-4`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M7-3-6	Using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files	`AUTOSAR C++14 Rule M7-3-6`
AUTOSAR C++14 Rule M7-4-2	Assembler instructions shall only be introduced using the asm declaration	`AUTOSAR C++14 Rule M7-4-2`
AUTOSAR C++14 Rule M7-4-3	Assembly language shall be encapsulated and isolated	`AUTOSAR C++14 Rule M7-4-3`
AUTOSAR C++14 Rule M7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function	`AUTOSAR C++14 Rule M7-5-1`
AUTOSAR C++14 Rule M7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist	`AUTOSAR C++14 Rule M7-5-2`
AUTOSAR C++14 Rule M8-0-1	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively	`AUTOSAR C++14 Rule M8-0-1`
AUTOSAR C++14 Rule M8-3-1	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments	`AUTOSAR C++14 Rule M8-3-1`
AUTOSAR C++14 Rule M8-4-2	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration	`AUTOSAR C++14 Rule M8-4-2`
AUTOSAR C++14 Rule M8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &	`AUTOSAR C++14 Rule M8-4-4`
AUTOSAR C++14 Rule M8-5-2	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures	`AUTOSAR C++14 Rule M8-5-2`
AUTOSAR C++14 Rule M9-3-1	`Const` member functions shall not return non-`const` pointers or references to class-data	`AUTOSAR C++14 Rule M9-3-1`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M9-3-3	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const	`AUTOSAR C++14 Rule M9-3-3`
AUTOSAR C++14 Rule M9-6-4	Named bit-fields with signed integer type shall have a length of more than one bit	`AUTOSAR C++14 Rule M9-6-4`

Unsupported Rules

Polyspace does not support these **Required** rules:

Rule	Description
M0-3-1	Minimization of run-time failures shall be ensured by the use of at least one of:\n (a) static analysis tools/techniques;\n (b) dynamic analysis tools/ techniques;\n (c) explicit coding of checks to handle run-time faults.
M0-4-1	Use of scaled-integer or fixed-point arithmetic shall be documented.
M0-4-2	Use of floating-point arithmetic shall be documented.
A0-4-1	Floating-point implementation shall comply with IEEE 754 standard.
A0-4-3	The implementations in the chosen compiler shall strictly comply with the C++14 Language Standard.
M1-0-2	Multiple compilers shall only be used if they have a common, defined interface.
A1-1-2	A warning level of the compilation process shall be set in compliance with project policies.
A1-1-3	An optimization option that disregards strict standard compliance shall not be turned on in the chosen compiler.
A1-2-1	When using a compiler toolchain (including preprocessor, compiler itself, linker, C++ standard libraries) in safety-related software, the tool confidence level (TCL) shall be determined. In case of TCL2 or TCL3, the compiler shall undergo a "Qualification of a software tool", as per ISO 26262-8.11.4.6 [5].
A1-4-1	Code metrics and their valid boundaries shall be defined and code shall comply with defined boundaries of code metrics.

Rule	Description
A2-7-5	Comments shall not document any actions or sources (e.g. tables, figures, paragraphs, etc.) that are outside of the file.
M5-17-1	The semantic equivalence between a binary operator and its assignment operator form shall be preserved.
M7-4-1	All usage of assembler shall be documented.
M9-6-1	When the absolute positioning of bits representing a bit-field is required, then the behavior and packing of bit-fields shall be documented.
A9-6-2	Bit-fields shall be used only when interfacing to hardware or conforming to communication protocols.
A10-0-1	Public inheritance shall be used to implement "is-a" relationship.
A10-0-2	Membership or non-public inheritance shall be used to implement "has-a" relationship.
A15-0-1	A function shall not exit with an exception if it is able to complete its task.
A15-0-4	Unchecked exceptions shall be used to represent errors from which the caller cannot reasonably be expected to recover.
A15-0-5	Checked exceptions shall be used to represent errors from which the caller can reasonably be expected to recover.
A15-0-6	An analysis shall be performed to analyze the failure modes of exception handling. In particular, the following failure modes shall be analyzed:\n (a) worst time execution time not existing or cannot be determined,\n (b) stack not correctly unwound,\n (c) exception not thrown, other exception thrown, wrong catch activated,\n (d) memory not available while exception handling.
A15-0-8	A worst-case execution time (WCET) analysis shall be performed to determine maximum execution time constraints of the software, covering in particular the exceptions processing.
A15-3-2	If a function throws an exception, it shall be handled when meaningful actions can be taken, otherwise it shall be propagated.
A17-0-2	All project's code including used libraries (including standard and user-defined libraries) and any third-party user code shall conform to the AUTOSAR C++14 Coding Guidelines.

Rule	Description
A18-5-6	An analysis shall be performed to analyze the failure modes of dynamic memory management. In particular, the following failure modes shall be analyzed:\n (a) non-deterministic behavior resulting with nonexistence of worst-case execution time,\n (b) memory fragmentation,\n (c) running out of memory,\n (d) mismatched allocations and deallocations,\n (e) dependence on non-deterministic calls to kernel.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

More About

- “Check for and Review Coding Standard Violations”
- “Coding Standards”
- “Statically Enforceable AUTOSAR C++14 Rules Supported by Polyspace Bug Finder”
- “Required MISRA C++:2008 Coding Rules Supported by Polyspace Bug Finder”
- “Checkers Deactivated in Polyspace as You Code Analysis”

Statically Enforceable AUTOSAR C++14 Rules Supported by Polyspace Bug Finder

The AUTOSAR C++14 standard classifies the rules that are statically enforceable as **Automated** and **Partially Automated**. In total, Polyspace supports 349 out of 349 [2]AUTOSAR C++14 coding rules that are enforceable by a static analysis tool.

Automated Rules

According to the AUTOSAR C++14 standard, static analysis detects all violations of the **Automated** rules. Polyspace Bug Finder supports 327 out of 327 **Automated** rules that can be enforced by a static analysis tool. The AUTOSAR C++14 standard contains two **Automated** rules that cannot be enforced by a static analysis tool.

Polyspace supports these **Automated** rules.

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A0-1-1	A project shall not contain instances of non-volatile variables being given values that are not subsequently used	`AUTOSAR C++14 Rule A0-1-1`
AUTOSAR C++14 Rule A0-1-2	The value returned by a function having a non-void return type that is not an overloaded operator shall be used	`AUTOSAR C++14 Rule A0-1-2`
AUTOSAR C++14 Rule A0-1-3	Every function defined in an anonymous namespace, or static function with internal linkage, or private member function shall be used	`AUTOSAR C++14 Rule A0-1-3`
AUTOSAR C++14 Rule A0-1-4	There shall be no unused named parameters in non-virtual functions	`AUTOSAR C++14 Rule A0-1-4`
AUTOSAR C++14 Rule A0-1-5	There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it	`AUTOSAR C++14 Rule A0-1-5`
AUTOSAR C++14 Rule A0-1-6	There should be no unused type declarations	`AUTOSAR C++14 Rule A0-1-6`
AUTOSAR C++14 Rule A0-4-2	Type long double shall not be used	`AUTOSAR C++14 Rule A0-4-2`

2 The AUTOSAR C++14 standard contains 351 statically enforceable rules. The rules A0-4-3 and A1-4-3 are not enforceable by a static analysis tool. These rules might be enforced by your compiler.

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A1-1-1	All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features	`AUTOSAR C++14 Rule A1-1-1`
AUTOSAR C++14 Rule A10-1-1	Class shall not be derived from more than one base class which is not an interface class	`AUTOSAR C++14 Rule A10-1-1`
AUTOSAR C++14 Rule A10-2-1	Non-virtual public or protected member functions shall not be redefined in derived classes	`AUTOSAR C++14 Rule A10-2-1`
AUTOSAR C++14 Rule A10-3-1	Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final	`AUTOSAR C++14 Rule A10-3-1`
AUTOSAR C++14 Rule A10-3-2	Each overriding virtual function shall be declared with the override or final specifier	`AUTOSAR C++14 Rule A10-3-2`
AUTOSAR C++14 Rule A10-3-3	Virtual functions shall not be introduced in a final class	`AUTOSAR C++14 Rule A10-3-3`
AUTOSAR C++14 Rule A10-3-5	A user-defined assignment operator shall not be virtual	`AUTOSAR C++14 Rule A10-3-5`
AUTOSAR C++14 Rule A11-0-1	A non-POD type should be defined as class	`AUTOSAR C++14 Rule A11-0-1`
AUTOSAR C++14 Rule A11-0-2	A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class	`AUTOSAR C++14 Rule A11-0-2`
AUTOSAR C++14 Rule A11-3-1	Friend declarations shall not be used	`AUTOSAR C++14 Rule A11-3-1`
AUTOSAR C++14 Rule A12-0-1	If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well	`AUTOSAR C++14 Rule A12-0-1`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A12-1-1	Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members	AUTOSAR C++14 Rule A12-1-1
AUTOSAR C++14 Rule A12-1-2	Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type	AUTOSAR C++14 Rule A12-1-2
AUTOSAR C++14 Rule A12-1-3	If all user-defined constructors of a class initialize data members with constant values that are the same across all constructors, then data members shall be initialized using NSDMI instead	AUTOSAR C++14 Rule A12-1-3
AUTOSAR C++14 Rule A12-1-4	All constructors that are callable with a single argument of fundamental type shall be declared explicit	AUTOSAR C++14 Rule A12-1-4
AUTOSAR C++14 Rule A12-1-6	Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors	AUTOSAR C++14 Rule A12-1-6
AUTOSAR C++14 Rule A12-4-1	Destructor of a base class shall be public virtual, public override or protected non-virtual	AUTOSAR C++14 Rule A12-4-1
AUTOSAR C++14 Rule A12-4-2	If a public destructor of a class is non-virtual, then the class should be declared final	AUTOSAR C++14 Rule A12-4-2
AUTOSAR C++14 Rule A12-6-1	All class data members that are initialized by the constructor shall be initialized using member initializers	AUTOSAR C++14 Rule A12-6-1
AUTOSAR C++14 Rule A12-7-1	If the behavior of a user-defined special member function is identical to implicitly defined special member function, then it shall be defined "=default" or be left undefined	AUTOSAR C++14 Rule A12-7-1
AUTOSAR C++14 Rule A12-8-1	Move and copy constructors shall move and respectively copy base classes and data members of a class, without any side effects	AUTOSAR C++14 Rule A12-8-1

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A12-8-2	User-defined copy and move assignment operators should use user-defined no-throw swap function	AUTOSAR C++14 Rule A12-8-2
AUTOSAR C++14 Rule A12-8-4	Move constructor shall not initialize its class members and base classes using copy semantics	AUTOSAR C++14 Rule A12-8-4
AUTOSAR C++14 Rule A12-8-5	A copy assignment and a move assignment operators shall handle self-assignment	AUTOSAR C++14 Rule A12-8-5
AUTOSAR C++14 Rule A12-8-6	Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class	AUTOSAR C++14 Rule A12-8-6
AUTOSAR C++14 Rule A12-8-7	Assignment operators should be declared with the ref-qualifier &	AUTOSAR C++14 Rule A12-8-7
AUTOSAR C++14 Rule A13-1-2	User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters	AUTOSAR C++14 Rule A13-1-2
AUTOSAR C++14 Rule A13-1-3	User defined literals operators shall only perform conversion of passed parameters	AUTOSAR C++14 Rule A13-1-3
AUTOSAR C++14 Rule A13-2-1	An assignment operator shall return a reference to "this"	AUTOSAR C++14 Rule A13-2-1
AUTOSAR C++14 Rule A13-2-2	A binary arithmetic operator and a bitwise operator shall return a "prvalue"	AUTOSAR C++14 Rule A13-2-2
AUTOSAR C++14 Rule A13-2-3	A relational operator shall return a boolean value	AUTOSAR C++14 Rule A13-2-3
AUTOSAR C++14 Rule A13-3-1	A function that contains "forwarding reference" as its argument shall not be overloaded	AUTOSAR C++14 Rule A13-3-1
AUTOSAR C++14 Rule A13-5-1	If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented	AUTOSAR C++14 Rule A13-5-1
AUTOSAR C++14 Rule A13-5-2	All user-defined conversion operators shall be defined explicit	AUTOSAR C++14 Rule A13-5-2
AUTOSAR C++14 Rule A13-5-3	User-defined conversion operators should not be used	AUTOSAR C++14 Rule A13-5-3

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A13-5-4	If two opposite operators are defined, one shall be defined in terms of the other	AUTOSAR C++14 Rule A13-5-4
AUTOSAR C++14 Rule A13-5-5	Comparison operators shall be non-member functions with identical parameter types and noexcept	AUTOSAR C++14 Rule A13-5-5
AUTOSAR C++14 Rule A13-6-1	Digit sequences separators ' shall only be used as follows: (1) for decimal, every 3 digits, (2) for hexadecimal, every 2 digits, (3) for binary, every 4 digits	AUTOSAR C++14 Rule A13-6-1
AUTOSAR C++14 Rule A14-5-1	A template constructor shall not participate in overload resolution for a single argument of the enclosing class type	AUTOSAR C++14 Rule A14-5-1
AUTOSAR C++14 Rule A14-5-3	A non-member generic operator shall only be declared in a namespace that does not contain class (struct) type, enum type or union type declarations	AUTOSAR C++14 Rule A14-5-3
AUTOSAR C++14 Rule A14-7-1	A type used as a template argument shall provide all members that are used by the template	AUTOSAR C++14 Rule A14-7-1
AUTOSAR C++14 Rule A14-7-2	Template specialization shall be declared in the same file (1) as the primary template (2) as a user-defined type, for which the specialization is declared	AUTOSAR C++14 Rule A14-7-2
AUTOSAR C++14 Rule A14-8-2	Explicit specializations of function templates shall not be used	AUTOSAR C++14 Rule A14-8-2
AUTOSAR C++14 Rule A15-1-1	Only instances of types derived from std::exception should be thrown	AUTOSAR C++14 Rule A15-1-1
AUTOSAR C++14 Rule A15-1-2	An exception object shall not be a pointer	AUTOSAR C++14 Rule A15-1-2
AUTOSAR C++14 Rule A15-1-3	All thrown exceptions should be unique	AUTOSAR C++14 Rule A15-1-3
AUTOSAR C++14 Rule A15-2-1	Constructors that are not noexcept shall not be invoked before program startup	AUTOSAR C++14 Rule A15-2-1

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A15-3-5	A class type exception shall be caught by reference or const reference	AUTOSAR C++14 Rule A15-3-5
AUTOSAR C++14 Rule A15-4-1	Dynamic exception-specification shall not be used	AUTOSAR C++14 Rule A15-4-1
AUTOSAR C++14 Rule A15-4-2	If a function is declared to be noexcept, noexcept(true) or noexcept(<true condition>), then it shall not exit with an exception	AUTOSAR C++14 Rule A15-4-2
AUTOSAR C++14 Rule A15-4-3	The noexcept specification of a function shall either be identical across all translation units, or identical or more restrictive between a virtual member function and an overrider	AUTOSAR C++14 Rule A15-4-3
AUTOSAR C++14 Rule A15-4-4	A declaration of non-throwing function shall contain noexcept specification	AUTOSAR C++14 Rule A15-4-4
AUTOSAR C++14 Rule A15-4-5	Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overriders	AUTOSAR C++14 Rule A15-4-5
AUTOSAR C++14 Rule A15-5-1	All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A noexcept exception specification shall be added to these functions as appropriate	AUTOSAR C++14 Rule A15-5-1
AUTOSAR C++14 Rule A15-5-3	The std::terminate() function shall not be called implicitly	AUTOSAR C++14 Rule A15-5-3
AUTOSAR C++14 Rule A16-0-1	The preprocessor shall only be used for unconditional and conditional file inclusion and include guards, and using specific directives	AUTOSAR C++14 Rule A16-0-1
AUTOSAR C++14 Rule A16-2-1	The ', ", /*, //, \ characters shall not occur in a header file name or in #include directive	AUTOSAR C++14 Rule A16-2-1

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A16-2-2	There shall be no unused include directives	AUTOSAR C++14 Rule A16-2-2
AUTOSAR C++14 Rule A16-6-1	#error directive shall not be used	AUTOSAR C++14 Rule A16-6-1
AUTOSAR C++14 Rule A16-7-1	The #pragma directive shall not be used	AUTOSAR C++14 Rule A16-7-1
AUTOSAR C++14 Rule A17-0-1	Reserved identifiers, macros and functions in the C++ standard library shall not be defined, redefined or undefined	AUTOSAR C++14 Rule A17-0-1
AUTOSAR C++14 Rule A17-6-1	Non-standard entities shall not be added to standard namespaces	AUTOSAR C++14 Rule A17-6-1
AUTOSAR C++14 Rule A18-0-1	The C library facilities shall only be accessed through C++ library headers	AUTOSAR C++14 Rule A18-0-1
AUTOSAR C++14 Rule A18-0-2	The error state of a conversion from string to a numeric value shall be checked	AUTOSAR C++14 Rule A18-0-2
AUTOSAR C++14 Rule A18-0-3	The library <clocale> (locale.h) and the setlocale function shall not be used	AUTOSAR C++14 Rule A18-0-3
AUTOSAR C++14 Rule A18-1-1	C-style arrays shall not be used	AUTOSAR C++14 Rule A18-1-1
AUTOSAR C++14 Rule A18-1-2	The std::vector<bool> specialization shall not be used	AUTOSAR C++14 Rule A18-1-2
AUTOSAR C++14 Rule A18-1-3	The std::auto_ptr shall not be used	AUTOSAR C++14 Rule A18-1-3
AUTOSAR C++14 Rule A18-1-4	A pointer pointing to an element of an array of objects shall not be passed to a smart pointer of single object type	AUTOSAR C++14 Rule A18-1-4
AUTOSAR C++14 Rule A18-1-6	All std::hash specializations for user-defined types shall have a noexcept function call operator	AUTOSAR C++14 Rule A18-1-6
AUTOSAR C++14 Rule A18-5-1	Functions malloc, calloc, realloc and free shall not be used	AUTOSAR C++14 Rule A18-5-1
AUTOSAR C++14 Rule A18-5-10	Placement new shall be used only with properly aligned pointers to sufficient storage capacity	AUTOSAR C++14 Rule A18-5-10
AUTOSAR C++14 Rule A18-5-11	"operator new" and "operator delete" shall be defined together	AUTOSAR C++14 Rule A18-5-11

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A18-5-3	The form of delete operator shall match the form of new operator used to allocate the memory	`AUTOSAR C++14 Rule A18-5-3`
AUTOSAR C++14 Rule A18-5-4	If a project has sized or unsized version of operator 'delete' globally defined, then both sized and unsized versions shall be defined	`AUTOSAR C++14 Rule A18-5-4`
AUTOSAR C++14 Rule A18-5-9	Custom implementations of dynamic memory allocation and deallocation functions shall meet the semantic requirements specified in the corresponding "Required behaviour" clause from the C++ Standard	`AUTOSAR C++14 Rule A18-5-9`
AUTOSAR C++14 Rule A18-9-1	The std::bind shall not be used	`AUTOSAR C++14 Rule A18-9-1`
AUTOSAR C++14 Rule A18-9-2	Forwarding values to other functions shall be done via: (1) std::move if the value is an rvalue reference, (2) std::forward if the value is forwarding reference	`AUTOSAR C++14 Rule A18-9-2`
AUTOSAR C++14 Rule A18-9-3	The std::move shall not be used on objects declared const or const&	`AUTOSAR C++14 Rule A18-9-3`
AUTOSAR C++14 Rule A18-9-4	An argument to std::forward shall not be subsequently used	`AUTOSAR C++14 Rule A18-9-4`
AUTOSAR C++14 Rule A2-10-1	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope	`AUTOSAR C++14 Rule A2-10-1`
AUTOSAR C++14 Rule A2-10-4	The identifier name of a non-member object with static storage duration or static function shall not be reused within a namespace	`AUTOSAR C++14 Rule A2-10-4`
AUTOSAR C++14 Rule A2-10-5	An identifier name of a function with static storage duration or a non-member object with external or internal linkage should not be reused	`AUTOSAR C++14 Rule A2-10-5`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A2-10-6	A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope	AUTOSAR C++14 Rule A2-10-6
AUTOSAR C++14 Rule A2-11-1	Volatile keyword shall not be used	AUTOSAR C++14 Rule A2-11-1
AUTOSAR C++14 Rule A2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2014 shall be used	AUTOSAR C++14 Rule A2-13-1
AUTOSAR C++14 Rule A2-13-2	String literals with different encoding prefixes shall not be concatenated	AUTOSAR C++14 Rule A2-13-2
AUTOSAR C++14 Rule A2-13-3	Type wchar_t shall not be used	AUTOSAR C++14 Rule A2-13-3
AUTOSAR C++14 Rule A2-13-4	String literals shall not be assigned to non-constant pointers	AUTOSAR C++14 Rule A2-13-4
AUTOSAR C++14 Rule A2-13-5	Hexadecimal constants should be uppercase	AUTOSAR C++14 Rule A2-13-5
AUTOSAR C++14 Rule A2-13-6	Universal character names shall be used only inside character or string literals	AUTOSAR C++14 Rule A2-13-6
AUTOSAR C++14 Rule A2-3-1	Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code	AUTOSAR C++14 Rule A2-3-1
AUTOSAR C++14 Rule A2-5-1	Trigraphs shall not be used	AUTOSAR C++14 Rule A2-5-1
AUTOSAR C++14 Rule A2-5-2	Digraphs shall not be used	AUTOSAR C++14 Rule A2-5-2
AUTOSAR C++14 Rule A2-7-1	The character \ shall not occur as a last character of a C++ comment	AUTOSAR C++14 Rule A2-7-1
AUTOSAR C++14 Rule A2-7-3	All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation	AUTOSAR C++14 Rule A2-7-3
AUTOSAR C++14 Rule A20-8-1	An already-owned pointer value shall not be stored in an unrelated smart pointer	AUTOSAR C++14 Rule A20-8-1
AUTOSAR C++14 Rule A20-8-2	A std::unique_ptr shall be used to represent exclusive ownership	AUTOSAR C++14 Rule A20-8-2

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A20-8-3	A std::shared_ptr shall be used to represent shared ownership	AUTOSAR C++14 Rule A20-8-3
AUTOSAR C++14 Rule A20-8-4	A std::unique_ptr shall be used over std::shared_ptr if ownership sharing is not required	AUTOSAR C++14 Rule A20-8-4
AUTOSAR C++14 Rule A20-8-5	std::make_unique shall be used to construct objects owned by std::unique_ptr	AUTOSAR C++14 Rule A20-8-5
AUTOSAR C++14 Rule A20-8-6	std::make_shared shall be used to construct objects owned by std::shared_ptr	AUTOSAR C++14 Rule A20-8-6
AUTOSAR C++14 Rule A21-8-1	Arguments to character-handling functions shall be representable as an unsigned char	AUTOSAR C++14 Rule A21-8-1
AUTOSAR C++14 Rule A23-0-1	An iterator shall not be implicitly converted to const_iterator	AUTOSAR C++14 Rule A23-0-1
AUTOSAR C++14 Rule A23-0-2	Elements of a container shall only be accessed via valid references, iterators, and pointers	AUTOSAR C++14 Rule A23-0-2
AUTOSAR C++14 Rule A25-1-1	Non-static data members or captured values of predicate function objects that are state related to this object's identity shall not be copied	AUTOSAR C++14 Rule A25-1-1
AUTOSAR C++14 Rule A26-5-1	Pseudorandom numbers shall not be generated using std::rand()	AUTOSAR C++14 Rule A26-5-1
AUTOSAR C++14 Rule A26-5-2	Random number engines shall not be default-initialized	AUTOSAR C++14 Rule A26-5-2
AUTOSAR C++14 Rule A27-0-2	A C-style string shall guarantee sufficient space for data and the null terminator	AUTOSAR C++14 Rule A27-0-2
AUTOSAR C++14 Rule A27-0-3	Alternate input and output operations on a file stream shall not be used without an intervening flush or positioning call	AUTOSAR C++14 Rule A27-0-3
AUTOSAR C++14 Rule A27-0-4	C-style strings shall not be used	AUTOSAR C++14 Rule A27-0-4

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A3-1-1	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule	AUTOSAR C++14 Rule A3-1-1
AUTOSAR C++14 Rule A3-1-2	Header files, that are defined locally in the project, shall have a file name extension of one of: .h, .hpp or .hxx	AUTOSAR C++14 Rule A3-1-2
AUTOSAR C++14 Rule A3-1-3	Implementation files, that are defined locally in the project, should have a file name extension of ".cpp"	AUTOSAR C++14 Rule A3-1-3
AUTOSAR C++14 Rule A3-1-4	When an array with external linkage is declared, its size shall be stated explicitly	AUTOSAR C++14 Rule A3-1-4
AUTOSAR C++14 Rule A3-1-6	Trivial accessor and mutator functions should be inlined	AUTOSAR C++14 Rule A3-1-6
AUTOSAR C++14 Rule A3-3-1	Objects or functions with external linkage (including members of named namespaces) shall be declared in a header file	AUTOSAR C++14 Rule A3-3-1
AUTOSAR C++14 Rule A3-3-2	Static and thread-local objects shall be constant-initialized	AUTOSAR C++14 Rule A3-3-2
AUTOSAR C++14 Rule A3-9-1	Fixed width integer types from <cstdint>, indicating the size and signedness, shall be used in place of the basic numerical types	AUTOSAR C++14 Rule A3-9-1
AUTOSAR C++14 Rule A4-10-1	Only nullptr literal shall be used as the null-pointer-constraint	AUTOSAR C++14 Rule A4-10-1
AUTOSAR C++14 Rule A4-5-1	Expressions with type enum or enum class shall not be used as operands to built-in and overloaded operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=	AUTOSAR C++14 Rule A4-5-1
AUTOSAR C++14 Rule A4-7-1	An integer expression shall not lead to data loss	AUTOSAR C++14 Rule A4-7-1

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits	AUTOSAR C++14 Rule A5-0-1
AUTOSAR C++14 Rule A5-0-2	The condition of an if-statement and the condition of an iteration statement shall have type bool	AUTOSAR C++14 Rule A5-0-2
AUTOSAR C++14 Rule A5-0-3	The declaration of objects shall contain no more than two levels of pointer indirection	AUTOSAR C++14 Rule A5-0-3
AUTOSAR C++14 Rule A5-0-4	Pointer arithmetic shall not be used with pointers to non-final classes	AUTOSAR C++14 Rule A5-0-4
AUTOSAR C++14 Rule A5-1-2	Variables shall not be implicitly captured in a lambda expression	AUTOSAR C++14 Rule A5-1-2
AUTOSAR C++14 Rule A5-1-3	Parameter list (possibly empty) shall be included in every lambda expression	AUTOSAR C++14 Rule A5-1-3
AUTOSAR C++14 Rule A5-1-4	A lambda expression object shall not outlive any of its reference-captured objects	AUTOSAR C++14 Rule A5-1-4
AUTOSAR C++14 Rule A5-1-6	Return type of a non-void return type lambda expression should be explicitly specified	AUTOSAR C++14 Rule A5-1-6
AUTOSAR C++14 Rule A5-1-7	A lambda shall not be an operand to decltype or typeid	AUTOSAR C++14 Rule A5-1-7
AUTOSAR C++14 Rule A5-1-8	Lambda expressions should not be defined inside another lambda expression	AUTOSAR C++14 Rule A5-1-8
AUTOSAR C++14 Rule A5-1-9	Identical unnamed lambda expressions shall be replaced with a named function or a named lambda expression	AUTOSAR C++14 Rule A5-1-9
AUTOSAR C++14 Rule A5-10-1	A pointer to member virtual function shall only be tested for equality with null-pointer-constant	AUTOSAR C++14 Rule A5-10-1
AUTOSAR C++14 Rule A5-16-1	The ternary conditional operator shall not be used as a sub-expression	AUTOSAR C++14 Rule A5-16-1
AUTOSAR C++14 Rule A5-2-1	dynamic_cast should not be used	AUTOSAR C++14 Rule A5-2-1
AUTOSAR C++14 Rule A5-2-2	Traditional C-style casts shall not be used	AUTOSAR C++14 Rule A5-2-2

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A5-2-3	A cast shall not remove any const or volatile qualification from the type of a pointer or reference	AUTOSAR C++14 Rule A5-2-3
AUTOSAR C++14 Rule A5-2-4	reinterpret_cast shall not be used	AUTOSAR C++14 Rule A5-2-4
AUTOSAR C++14 Rule A5-2-5	An array or container shall not be accessed beyond its range	AUTOSAR C++14 Rule A5-2-5
AUTOSAR C++14 Rule A5-2-6	The operands of a logical && or `||` shall be parenthesized if the operands contain binary operators	AUTOSAR C++14 Rule A5-2-6
AUTOSAR C++14 Rule A5-3-3	Pointers to incomplete class types shall not be deleted	AUTOSAR C++14 Rule A5-3-3
AUTOSAR C++14 Rule A5-5-1	A pointer to member shall not access non-existent class members	AUTOSAR C++14 Rule A5-5-1
AUTOSAR C++14 Rule A5-6-1	The right hand operand of the integer division or remainder operators shall not be equal to zero	AUTOSAR C++14 Rule A5-6-1
AUTOSAR C++14 Rule A6-2-1	Move and copy assignment operators shall either move or respectively copy base classes and data members of a class, without any side effects	AUTOSAR C++14 Rule A6-2-1
AUTOSAR C++14 Rule A6-2-2	Expression statements shall not be explicit calls to constructors of temporary objects only	AUTOSAR C++14 Rule A6-2-2
AUTOSAR C++14 Rule A6-4-1	A `switch` statement shall have at least two case-clauses, distinct from the default label	AUTOSAR C++14 Rule A6-4-1
AUTOSAR C++14 Rule A6-5-1	A for-loop that loops through all elements of the container and does not use its loop-counter shall not be used	AUTOSAR C++14 Rule A6-5-1
AUTOSAR C++14 Rule A6-5-2	A `for` loop shall contain a single loop-counter which shall not have floating-point type	AUTOSAR C++14 Rule A6-5-2
AUTOSAR C++14 Rule A6-5-3	Do statements should not be used	AUTOSAR C++14 Rule A6-5-3
AUTOSAR C++14 Rule A6-5-4	For-init-statement and expression should not perform actions other than loop-counter initialization and modification	AUTOSAR C++14 Rule A6-5-4

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A6-6-1	The `goto` statement shall not be used	`AUTOSAR C++14 Rule A6-6-1`
AUTOSAR C++14 Rule A7-1-1	Constexpr or const specifiers shall be used for immutable data declaration	`AUTOSAR C++14 Rule A7-1-1`
AUTOSAR C++14 Rule A7-1-2	The constexpr specifier shall be used for values that can be determined at compile time	`AUTOSAR C++14 Rule A7-1-2`
AUTOSAR C++14 Rule A7-1-3	CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name	`AUTOSAR C++14 Rule A7-1-3`
AUTOSAR C++14 Rule A7-1-4	The register keyword shall not be used	`AUTOSAR C++14 Rule A7-1-4`
AUTOSAR C++14 Rule A7-1-5	The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax	`AUTOSAR C++14 Rule A7-1-5`
AUTOSAR C++14 Rule A7-1-6	The typedef specifier shall not be used	`AUTOSAR C++14 Rule A7-1-6`
AUTOSAR C++14 Rule A7-1-7	Each expression statement and identifier declaration shall be placed on a separate line	`AUTOSAR C++14 Rule A7-1-7`
AUTOSAR C++14 Rule A7-1-8	A non-type specifier shall be placed before a type specifier in a declaration	`AUTOSAR C++14 Rule A7-1-8`
AUTOSAR C++14 Rule A7-1-9	A class, structure, or enumeration shall not be declared in the definition of its type	`AUTOSAR C++14 Rule A7-1-9`
AUTOSAR C++14 Rule A7-2-1	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration	`AUTOSAR C++14 Rule A7-2-1`
AUTOSAR C++14 Rule A7-2-2	Enumeration underlying type shall be explicitly defined	`AUTOSAR C++14 Rule A7-2-2`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A7-2-3	Enumerations shall be declared as scoped enum classes	AUTOSAR C++14 Rule A7-2-3
AUTOSAR C++14 Rule A7-2-4	In an enumeration, either (1) none, (2) the first or (3) all enumerators shall be initialized	AUTOSAR C++14 Rule A7-2-4
AUTOSAR C++14 Rule A7-3-1	All overloads of a function shall be visible from where it is called	AUTOSAR C++14 Rule A7-3-1
AUTOSAR C++14 Rule A7-4-1	The asm declaration shall not be used	AUTOSAR C++14 Rule A7-4-1
AUTOSAR C++14 Rule A7-5-1	A function shall not return a reference or a pointer to a parameter that is passed by reference to const	AUTOSAR C++14 Rule A7-5-1
AUTOSAR C++14 Rule A7-5-2	Functions shall not call themselves, either directly or indirectly	AUTOSAR C++14 Rule A7-5-2
AUTOSAR C++14 Rule A7-6-1	Functions declared with the [[noreturn]] attribute shall not return	AUTOSAR C++14 Rule A7-6-1
AUTOSAR C++14 Rule A8-2-1	When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters	AUTOSAR C++14 Rule A8-2-1
AUTOSAR C++14 Rule A8-4-1	Functions shall not be defined using the ellipsis notation	AUTOSAR C++14 Rule A8-4-1
AUTOSAR C++14 Rule A8-4-10	A parameter shall be passed by reference if it can't be NULL	AUTOSAR C++14 Rule A8-4-10
AUTOSAR C++14 Rule A8-4-11	A smart pointer shall only be used as a parameter type if it expresses lifetime semantics	AUTOSAR C++14 Rule A8-4-11
AUTOSAR C++14 Rule A8-4-12	A std::unique_ptr shall be passed to a function as: (1) a copy to express the function assumes ownership (2) an lvalue reference to express that the function replaces the managed object.	AUTOSAR C++14 Rule A8-4-12

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A8-4-13	A std::shared_ptr shall be passed to a function as: (1) a copy to express the function shares ownership (2) an lvalue reference to express that the function replaces the managed object (3) a `const` lvalue reference to express that the function retains a reference count.	`AUTOSAR C++14 Rule A8-4-13`
AUTOSAR C++14 Rule A8-4-2	All exit paths from a function with non-void return type shall have an explicit return statement with an expression	`AUTOSAR C++14 Rule A8-4-2`
AUTOSAR C++14 Rule A8-4-4	Multiple output values from a function should be returned as a struct or tuple	`AUTOSAR C++14 Rule A8-4-4`
AUTOSAR C++14 Rule A8-4-5	"consume" parameters declared as X && shall always be moved from	`AUTOSAR C++14 Rule A8-4-5`
AUTOSAR C++14 Rule A8-4-6	"forward" parameters declared as T && shall always be forwarded	`AUTOSAR C++14 Rule A8-4-6`
AUTOSAR C++14 Rule A8-4-7	"in" parameters for "cheap to copy" types shall be passed by value	`AUTOSAR C++14 Rule A8-4-7`
AUTOSAR C++14 Rule A8-4-8	Output parameters shall not be used	`AUTOSAR C++14 Rule A8-4-8`
AUTOSAR C++14 Rule A8-4-9	"in-out" parameters declared as T & shall be modified	`AUTOSAR C++14 Rule A8-4-9`
AUTOSAR C++14 Rule A8-5-0	All memory shall be initialized before it is read	`AUTOSAR C++14 Rule A8-5-0`
AUTOSAR C++14 Rule A8-5-1	In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition	`AUTOSAR C++14 Rule A8-5-1`
AUTOSAR C++14 Rule A8-5-2	Braced-initialization {}, without equals sign, shall be used for variable initialization	`AUTOSAR C++14 Rule A8-5-2`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A8-5-3	A variable of type auto shall not be initialized using {} or ={} braced-initialization	AUTOSAR C++14 Rule A8-5-3
AUTOSAR C++14 Rule A8-5-4	If a class has a user-declared constructor that takes a parameter of type std::initializer_list, then it shall be the only constructor apart from special member function constructors	AUTOSAR C++14 Rule A8-5-4
AUTOSAR C++14 Rule A9-5-1	Unions shall not be used	AUTOSAR C++14 Rule A9-5-1
AUTOSAR C++14 Rule M0-1-1	A project shall not contain unreachable code	AUTOSAR C++14 Rule M0-1-1
AUTOSAR C++14 Rule M0-1-10	Every defined function should be called at least once	AUTOSAR C++14 Rule M0-1-10
AUTOSAR C++14 Rule M0-1-2	A project shall not contain infeasible paths	AUTOSAR C++14 Rule M0-1-2
AUTOSAR C++14 Rule M0-1-3	A project shall not contain unused variables	AUTOSAR C++14 Rule M0-1-3
AUTOSAR C++14 Rule M0-1-4	A project shall not contain non-volatile POD variables having only one use	AUTOSAR C++14 Rule M0-1-4
AUTOSAR C++14 Rule M0-1-8	All functions with void return type shall have external side effect(s)	AUTOSAR C++14 Rule M0-1-8
AUTOSAR C++14 Rule M0-1-9	There shall be no dead code	AUTOSAR C++14 Rule M0-1-9
AUTOSAR C++14 Rule M0-2-1	An object shall not be assigned to an overlapping object	AUTOSAR C++14 Rule M0-2-1
AUTOSAR C++14 Rule M10-1-1	Classes should not be derived from virtual bases	AUTOSAR C++14 Rule M10-1-1
AUTOSAR C++14 Rule M10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy	AUTOSAR C++14 Rule M10-1-2
AUTOSAR C++14 Rule M10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy	AUTOSAR C++14 Rule M10-1-3
AUTOSAR C++14 Rule M10-2-1	All accessible entity names within a multiple inheritance hierarchy should be unique	AUTOSAR C++14 Rule M10-2-1

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual	`AUTOSAR C++14 Rule M10-3-3`
AUTOSAR C++14 Rule M11-0-1	Member data in non-POD class types shall be private	`AUTOSAR C++14 Rule M11-0-1`
AUTOSAR C++14 Rule M12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor	`AUTOSAR C++14 Rule M12-1-1`
AUTOSAR C++14 Rule M14-5-3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter	`AUTOSAR C++14 Rule M14-5-3`
AUTOSAR C++14 Rule M14-6-1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->	`AUTOSAR C++14 Rule M14-6-1`
AUTOSAR C++14 Rule M15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement	`AUTOSAR C++14 Rule M15-0-3`
AUTOSAR C++14 Rule M15-1-1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown	`AUTOSAR C++14 Rule M15-1-1`
AUTOSAR C++14 Rule M15-1-2	NULL shall not be thrown explicitly	`AUTOSAR C++14 Rule M15-1-2`
AUTOSAR C++14 Rule M15-1-3	An empty throw (throw;) shall only be used in the compound statement of a catch handler	`AUTOSAR C++14 Rule M15-1-3`
AUTOSAR C++14 Rule M15-3-1	Exceptions shall be raised only after start-up and before termination	`AUTOSAR C++14 Rule M15-3-1`
AUTOSAR C++14 Rule M15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases	`AUTOSAR C++14 Rule M15-3-3`
AUTOSAR C++14 Rule M15-3-4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point	`AUTOSAR C++14 Rule M15-3-4`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class	`AUTOSAR C++14 Rule M15-3-6`
AUTOSAR C++14 Rule M15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last	`AUTOSAR C++14 Rule M15-3-7`
AUTOSAR C++14 Rule M16-0-1	#include directives in a file shall only be preceded by other preprocessor directives or comments	`AUTOSAR C++14 Rule M16-0-1`
AUTOSAR C++14 Rule M16-0-2	Macros shall only be #define'd or #undef'd in the global namespace	`AUTOSAR C++14 Rule M16-0-2`
AUTOSAR C++14 Rule M16-0-5	Arguments to a function-like macro shall not contain tokens that look like pre-processing directives	`AUTOSAR C++14 Rule M16-0-5`
AUTOSAR C++14 Rule M16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##	`AUTOSAR C++14 Rule M16-0-6`
AUTOSAR C++14 Rule M16-0-7	Undefined macro identifiers shall not be used in #if or #elif pre-processor directives, except as operands to the defined operator	`AUTOSAR C++14 Rule M16-0-7`
AUTOSAR C++14 Rule M16-0-8	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token	`AUTOSAR C++14 Rule M16-0-8`
AUTOSAR C++14 Rule M16-1-1	The defined pre-processor operator shall only be used in one of the two standard forms	`AUTOSAR C++14 Rule M16-1-1`
AUTOSAR C++14 Rule M16-1-2	All #else, #elif and #endif pre-processor directives shall reside in the same file as the #if or #ifdef directive to which they are related	`AUTOSAR C++14 Rule M16-1-2`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M16-2-3	Include guards shall be provided	AUTOSAR C++14 Rule M16-2-3
AUTOSAR C++14 Rule M16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition	AUTOSAR C++14 Rule M16-3-1
AUTOSAR C++14 Rule M16-3-2	The # and ## operators should not be used	AUTOSAR C++14 Rule M16-3-2
AUTOSAR C++14 Rule M17-0-2	The names of standard library macros and objects shall not be reused	AUTOSAR C++14 Rule M17-0-2
AUTOSAR C++14 Rule M17-0-3	The names of standard library functions shall not be overridden	AUTOSAR C++14 Rule M17-0-3
AUTOSAR C++14 Rule M17-0-5	The setjmp macro and the longjmp function shall not be used	AUTOSAR C++14 Rule M17-0-5
AUTOSAR C++14 Rule M18-0-3	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used	AUTOSAR C++14 Rule M18-0-3
AUTOSAR C++14 Rule M18-0-4	The time handling functions of library <ctime> shall not be used	AUTOSAR C++14 Rule M18-0-4
AUTOSAR C++14 Rule M18-0-5	The unbounded functions of library <cstring> shall not be used	AUTOSAR C++14 Rule M18-0-5
AUTOSAR C++14 Rule M18-2-1	The macro offsetof shall not be used	AUTOSAR C++14 Rule M18-2-1
AUTOSAR C++14 Rule M18-7-1	The signal handling facilities of <csignal> shall not be used	AUTOSAR C++14 Rule M18-7-1
AUTOSAR C++14 Rule M19-3-1	The error indicator errno shall not be used	AUTOSAR C++14 Rule M19-3-1
AUTOSAR C++14 Rule M2-10-1	Different identifiers shall be typographically unambiguous	AUTOSAR C++14 Rule M2-10-1
AUTOSAR C++14 Rule M2-13-2	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used	AUTOSAR C++14 Rule M2-13-2
AUTOSAR C++14 Rule M2-13-3	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type	AUTOSAR C++14 Rule M2-13-3
AUTOSAR C++14 Rule M2-13-4	Literal suffixes shall be upper case	AUTOSAR C++14 Rule M2-13-4

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M2-7-1	The character sequence /* shall not be used within a C-style comment	`AUTOSAR C++14 Rule M2-7-1`
AUTOSAR C++14 Rule M27-0-1	The stream input/output library <cstdio> shall not be used	`AUTOSAR C++14 Rule M27-0-1`
AUTOSAR C++14 Rule M3-1-2	Functions shall not be declared at block scope	`AUTOSAR C++14 Rule M3-1-2`
AUTOSAR C++14 Rule M3-2-1	All declarations of an object or function shall have compatible types	`AUTOSAR C++14 Rule M3-2-1`
AUTOSAR C++14 Rule M3-2-2	The One Definition Rule shall not be violated	`AUTOSAR C++14 Rule M3-2-2`
AUTOSAR C++14 Rule M3-2-3	A type, object or function that is used in multiple translation units shall be declared in one and only one file	`AUTOSAR C++14 Rule M3-2-3`
AUTOSAR C++14 Rule M3-2-4	An identifier with external linkage shall have exactly one definition	`AUTOSAR C++14 Rule M3-2-4`
AUTOSAR C++14 Rule M3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier	`AUTOSAR C++14 Rule M3-3-2`
AUTOSAR C++14 Rule M3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility	`AUTOSAR C++14 Rule M3-4-1`
AUTOSAR C++14 Rule M3-9-1	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations	`AUTOSAR C++14 Rule M3-9-1`
AUTOSAR C++14 Rule M3-9-3	The underlying bit representations of floating-point values shall not be used	`AUTOSAR C++14 Rule M3-9-3`
AUTOSAR C++14 Rule M4-10-1	NULL shall not be used as an integer value	`AUTOSAR C++14 Rule M4-10-1`
AUTOSAR C++14 Rule M4-10-2	Literal zero (0) shall not be used as the null-pointer-constant	`AUTOSAR C++14 Rule M4-10-2`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and ! =, the unary & operator, and the conditional operator	`AUTOSAR C++14 Rule M4-5-1`
AUTOSAR C++14 Rule M4-5-3	Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and ! =, and the unary & operator	`AUTOSAR C++14 Rule M4-5-3`
AUTOSAR C++14 Rule M5-0-10	If the bitwise operators ~and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand	`AUTOSAR C++14 Rule M5-0-10`
AUTOSAR C++14 Rule M5-0-11	The plain char type shall only be used for the storage and use of character values	`AUTOSAR C++14 Rule M5-0-11`
AUTOSAR C++14 Rule M5-0-12	Signed char and unsigned char type shall only be used for the storage and use of numeric values	`AUTOSAR C++14 Rule M5-0-12`
AUTOSAR C++14 Rule M5-0-14	The first operand of a conditional-operator shall have type bool	`AUTOSAR C++14 Rule M5-0-14`
AUTOSAR C++14 Rule M5-0-15	Array indexing shall be the only form of pointer arithmetic	`AUTOSAR C++14 Rule M5-0-15`
AUTOSAR C++14 Rule M5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array	`AUTOSAR C++14 Rule M5-0-16`
AUTOSAR C++14 Rule M5-0-17	Subtraction between pointers shall only be applied to pointers that address elements of the same array	`AUTOSAR C++14 Rule M5-0-17`
AUTOSAR C++14 Rule M5-0-18	`>`, `>=`, `<`, `<=` shall not be applied to objects of pointer type, except where they point to the same array	`AUTOSAR C++14 Rule M5-0-18`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M5-0-20	Non-constant operands to a binary bitwise operator shall have the same underlying type	`AUTOSAR C++14 Rule M5-0-20`
AUTOSAR C++14 Rule M5-0-21	Bitwise operators shall only be applied to operands of unsigned underlying type	`AUTOSAR C++14 Rule M5-0-21`
AUTOSAR C++14 Rule M5-0-3	A cvalue expression shall not be implicitly converted to a different underlying type	`AUTOSAR C++14 Rule M5-0-3`
AUTOSAR C++14 Rule M5-0-4	An implicit integral conversion shall not change the signedness of the underlying type	`AUTOSAR C++14 Rule M5-0-4`
AUTOSAR C++14 Rule M5-0-5	There shall be no implicit floating-integral conversions	`AUTOSAR C++14 Rule M5-0-5`
AUTOSAR C++14 Rule M5-0-6	An implicit integral or floating-point conversion shall not reduce the size of the underlying type	`AUTOSAR C++14 Rule M5-0-6`
AUTOSAR C++14 Rule M5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression	`AUTOSAR C++14 Rule M5-0-7`
AUTOSAR C++14 Rule M5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression	`AUTOSAR C++14 Rule M5-0-8`
AUTOSAR C++14 Rule M5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression	`AUTOSAR C++14 Rule M5-0-9`
AUTOSAR C++14 Rule M5-14-1	The right hand operand of a logical &&, || operators shall not contain side effects	`AUTOSAR C++14 Rule M5-14-1`
AUTOSAR C++14 Rule M5-18-1	The comma operator shall not be used	`AUTOSAR C++14 Rule M5-18-1`
AUTOSAR C++14 Rule M5-19-1	Evaluation of constant unsigned integer expressions shall not lead to wrap-around	`AUTOSAR C++14 Rule M5-19-1`
AUTOSAR C++14 Rule M5-2-10	The increment (++) and decrement (--) operators shall not be mixed with other operators in an expression	`AUTOSAR C++14 Rule M5-2-10`
AUTOSAR C++14 Rule M5-2-11	The comma operator, && operator and the || operator shall not be overloaded	`AUTOSAR C++14 Rule M5-2-11`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M5-2-12	An identifier with array type passed as a function argument shall not decay to a pointer	`AUTOSAR C++14 Rule M5-2-12`
AUTOSAR C++14 Rule M5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of `dynamic_cast`	`AUTOSAR C++14 Rule M5-2-2`
AUTOSAR C++14 Rule M5-2-3	Casts from a base class to a derived class should not be performed on polymorphic types	`AUTOSAR C++14 Rule M5-2-3`
AUTOSAR C++14 Rule M5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type	`AUTOSAR C++14 Rule M5-2-6`
AUTOSAR C++14 Rule M5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type	`AUTOSAR C++14 Rule M5-2-8`
AUTOSAR C++14 Rule M5-2-9	A cast shall not convert a pointer type to an integral type	`AUTOSAR C++14 Rule M5-2-9`
AUTOSAR C++14 Rule M5-3-1	Each operand of the ! operator, the logical && or the logical || operators shall have type bool	`AUTOSAR C++14 Rule M5-3-1`
AUTOSAR C++14 Rule M5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned	`AUTOSAR C++14 Rule M5-3-2`
AUTOSAR C++14 Rule M5-3-3	The unary & operator shall not be overloaded	`AUTOSAR C++14 Rule M5-3-3`
AUTOSAR C++14 Rule M5-3-4	Evaluation of the operand to the sizeof operator shall not contain side effects	`AUTOSAR C++14 Rule M5-3-4`
AUTOSAR C++14 Rule M6-2-1	Assignment operators shall not be used in sub-expressions	`AUTOSAR C++14 Rule M6-2-1`
AUTOSAR C++14 Rule M6-2-3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character	`AUTOSAR C++14 Rule M6-2-3`
AUTOSAR C++14 Rule M6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement	`AUTOSAR C++14 Rule M6-3-1`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M6-4-1	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement	`AUTOSAR C++14 Rule M6-4-1`
AUTOSAR C++14 Rule M6-4-2	All if ... else if constructs shall be terminated with an else clause	`AUTOSAR C++14 Rule M6-4-2`
AUTOSAR C++14 Rule M6-4-3	A switch statement shall be a well-formed switch statement	`AUTOSAR C++14 Rule M6-4-3`
AUTOSAR C++14 Rule M6-4-4	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	`AUTOSAR C++14 Rule M6-4-4`
AUTOSAR C++14 Rule M6-4-5	An unconditional throw or break statement shall terminate every non-empty switch-clause	`AUTOSAR C++14 Rule M6-4-5`
AUTOSAR C++14 Rule M6-4-6	The final clause of a `switch` statement shall be the default-clause	`AUTOSAR C++14 Rule M6-4-6`
AUTOSAR C++14 Rule M6-4-7	The condition of a switch statement shall not have bool type	`AUTOSAR C++14 Rule M6-4-7`
AUTOSAR C++14 Rule M6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=	`AUTOSAR C++14 Rule M6-5-2`
AUTOSAR C++14 Rule M6-5-3	The loop-counter shall not be modified within condition or statement	`AUTOSAR C++14 Rule M6-5-3`
AUTOSAR C++14 Rule M6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop	`AUTOSAR C++14 Rule M6-5-4`
AUTOSAR C++14 Rule M6-5-5	A loop-control-variable other than the loop-counter shall not be modified within condition or expression	`AUTOSAR C++14 Rule M6-5-5`
AUTOSAR C++14 Rule M6-5-6	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool	`AUTOSAR C++14 Rule M6-5-6`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement	`AUTOSAR C++14 Rule M6-6-1`
AUTOSAR C++14 Rule M6-6-2	The goto statement shall jump to a label declared later in the same function body	`AUTOSAR C++14 Rule M6-6-2`
AUTOSAR C++14 Rule M6-6-3	The `continue` statement shall only be used within a well-formed `for` loop	`AUTOSAR C++14 Rule M6-6-3`
AUTOSAR C++14 Rule M7-1-2	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified	`AUTOSAR C++14 Rule M7-1-2`
AUTOSAR C++14 Rule M7-3-1	The global namespace shall only contain main, namespace declarations and extern "C" declarations	`AUTOSAR C++14 Rule M7-3-1`
AUTOSAR C++14 Rule M7-3-2	The identifier main shall not be used for a function other than the global function main	`AUTOSAR C++14 Rule M7-3-2`
AUTOSAR C++14 Rule M7-3-3	There shall be no unnamed namespaces in header files	`AUTOSAR C++14 Rule M7-3-3`
AUTOSAR C++14 Rule M7-3-4	Using-directives shall not be used	`AUTOSAR C++14 Rule M7-3-4`
AUTOSAR C++14 Rule M7-3-6	Using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files	`AUTOSAR C++14 Rule M7-3-6`
AUTOSAR C++14 Rule M7-4-2	Assembler instructions shall only be introduced using the asm declaration	`AUTOSAR C++14 Rule M7-4-2`
AUTOSAR C++14 Rule M7-4-3	Assembly language shall be encapsulated and isolated	`AUTOSAR C++14 Rule M7-4-3`
AUTOSAR C++14 Rule M8-0-1	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively	`AUTOSAR C++14 Rule M8-0-1`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M8-3-1	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments	`AUTOSAR C++14 Rule M8-3-1`
AUTOSAR C++14 Rule M8-4-2	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration	`AUTOSAR C++14 Rule M8-4-2`
AUTOSAR C++14 Rule M8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &	`AUTOSAR C++14 Rule M8-4-4`
AUTOSAR C++14 Rule M8-5-2	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures	`AUTOSAR C++14 Rule M8-5-2`
AUTOSAR C++14 Rule M9-3-1	`Const` member functions shall not return non-`const` pointers or references to class-data	`AUTOSAR C++14 Rule M9-3-1`
AUTOSAR C++14 Rule M9-3-3	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const	`AUTOSAR C++14 Rule M9-3-3`
AUTOSAR C++14 Rule M9-6-4	Named bit-fields with signed integer type shall have a length of more than one bit	`AUTOSAR C++14 Rule M9-6-4`

Partially Automated Rules

According to the AUTOSAR C++14 standard, static analysis detects only a subset of all possible violation of **Partially Automated** rules. Polyspace Bug Finder supports 22 out of 22 **Partially Automated** rules. For details about which error scenarios of a rule Polyspace detects, see the **Polyspace Implementation** section in the reference page of the rule.

Polyspace supports these **Partially Automated** rules.

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A0-4-4	Range, domain and pole errors shall be checked when using math functions	`AUTOSAR C++14 Rule A0-4-4`
AUTOSAR C++14 Rule A12-0-2	Bitwise operations and operations that assume data representation in memory shall not be performed on objects	`AUTOSAR C++14 Rule A12-0-2`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A12-1-5	Common class initialization for non-constant members shall be done by a delegating constructor	AUTOSAR C++14 Rule A12-1-5
AUTOSAR C++14 Rule A12-8-3	Moved-from object shall not be read-accessed	AUTOSAR C++14 Rule A12-8-3
AUTOSAR C++14 Rule A14-5-2	Class members that are not dependent on template class parameters should be defined in a separate base class	AUTOSAR C++14 Rule A14-5-2
AUTOSAR C++14 Rule A15-0-2	At least the basic guarantee for exception safety shall be provided for all operations. In addition, each function may offer either the strong guarantee or the nothrow guarantee	AUTOSAR C++14 Rule A15-0-2
AUTOSAR C++14 Rule A15-0-7	Exception handling mechanism shall guarantee a deterministic worst-case time execution time	AUTOSAR C++14 Rule A15-0-7
AUTOSAR C++14 Rule A15-1-4	If a function exits with an exception, then before a throw, the function shall place all objects/resources that the function constructed in valid states or it shall delete them.	AUTOSAR C++14 Rule A15-1-4
AUTOSAR C++14 Rule A15-2-2	If a constructor is not noexcept and the constructor cannot finish object initialization, then it shall deallocate the object's resources and it shall throw an exception	AUTOSAR C++14 Rule A15-2-2
AUTOSAR C++14 Rule A15-3-3	Main function and a task main function shall catch at least: base class exceptions from all third-party libraries used, std::exception and all otherwise unhandled exceptions	AUTOSAR C++14 Rule A15-3-3
AUTOSAR C++14 Rule A15-5-2	Program shall not be abruptly terminated. In particular, an implicit or explicit invocation of std::abort(), std::quick_exit(), std::_Exit(), std::terminate() shall not be done	AUTOSAR C++14 Rule A15-5-2
AUTOSAR C++14 Rule A18-5-2	Non-placement new or delete expressions shall not be used	AUTOSAR C++14 Rule A18-5-2

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule A18-5-5	Memory management functions shall ensure the following: (a) deterministic behavior resulting with the existence of worst-case execution time, (b) avoiding memory fragmentation, (c) avoid running out of memory, (d) avoiding mismatched allocations or deallocations, (e) no dependence on non-deterministic calls to kernel	`AUTOSAR C++14 Rule A18-5-5`
AUTOSAR C++14 Rule A18-5-8	Objects that do not outlive a function shall have automatic storage duration	`AUTOSAR C++14 Rule A18-5-8`
AUTOSAR C++14 Rule A3-1-5	A function definition shall only be placed in a class definition if (1) the function is intended to be inlined (2) it is a member function template (3) it is a member function of a class template	`AUTOSAR C++14 Rule A3-1-5`
AUTOSAR C++14 Rule A5-1-1	Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead	`AUTOSAR C++14 Rule A5-1-1`
AUTOSAR C++14 Rule A5-3-2	Null pointers shall not be dereferenced	`AUTOSAR C++14 Rule A5-3-2`
AUTOSAR C++14 Rule A9-3-1	Member functions shall not return non-constant "raw" pointers or references to private or protected data owned by the class	`AUTOSAR C++14 Rule A9-3-1`
AUTOSAR C++14 Rule A9-6-1	Data types used for interfacing with hardware or conforming to communication protocols shall be trivial, standard-layout and only contain members of types with defined sizes	`AUTOSAR C++14 Rule A9-6-1`
AUTOSAR C++14 Rule M5-0-2	Limited dependence should be placed on C++ operator precedence rules in expressions	`AUTOSAR C++14 Rule M5-0-2`
AUTOSAR C++14 Rule M5-8-1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand	`AUTOSAR C++14 Rule M5-8-1`

AUTOSAR C++14 Rule	Description	Polyspace Checker
AUTOSAR C++14 Rule M6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality	AUTOSAR C++14 Rule M6-2-2

See Also

Check AUTOSAR C++ 14 (-autosar-cpp14)

More About

- "Check for and Review Coding Standard Violations"
- "Coding Standards"
- "Required AUTOSAR C++14 Coding Rules Supported by Polyspace Bug Finder"
- "Required MISRA C++:2008 Coding Rules Supported by Polyspace Bug Finder"
- "Checkers Deactivated in Polyspace as You Code Analysis"

Configure Verification of Modules or Libraries

- “Provide Context for C Code Verification” on page 18-2
- “Provide Context for C++ Code Verification” on page 18-4
- “Verify C Application Without main Function” on page 18-6
- “Verify C++ Classes” on page 18-9
- “Constrain Variable Ranges for Polyspace Analysis Using Manual Stubs and Manual main() Function” on page 18-18

Provide Context for C Code Verification

This example shows how to provide context for your C code verification. If you use default options and do not provide a `main` function, Polyspace Code Prover checks your code for robustness against all verification conditions. For instance, the software:

- Considers that global variables and inputs of uncalled functions are full range.
- Generates a `main` that calls uncalled functions in arbitrary order.

In addition, if you do not define a function but declare and call it in your code, Polyspace stubs the function. For a detailed list of assumptions, see "Code Prover Analysis Assumptions".

You can use analysis options on the **Configuration** pane to change the default behavior and provide more context about your code. Performing contextual verification can result in more proven code and therefore fewer orange checks.

Control Variable Range

Use the following options. In the user interface of the Polyspace desktop products, the options appear under the **Code Prover Verification** node.

Option	Purpose
`Variables to initialize (-main-generator-writes-variables)`	Specify the global variables that Polyspace must consider as initialized despite no explicit initialization in the code.
`Constraint setup (-data-range-specifications)`	Specify range for global variables.

Control Function Call Sequence

Use the following options. In the user interface of the Polyspace desktop products, the options appear under the **Code Prover Verification** node.

Option	Purpose
`Initialization functions (-functions-called-before-main)`	Specify the functions that the generated `main` must call first.
`Functions to call (-main-generator-calls)`	Specify the functions that the generated `main` must call later.

Control Stubbing Behavior

Use the following options. In the user interface of the Polyspace desktop products, the options appear under the **Inputs & Stubbing** node.

Option	Purpose
`Functions to stub (-functions-to-stub)`	Specify the functions that Polyspace must stub.

See Also

More About

- "Verify C Application Without main Function" on page 18-6

Provide Context for C++ Code Verification

This example shows how to provide context to your C++ code verification. If you use default options and do not provide a `main` function, Polyspace Code Prover checks your code for robustness against all verification conditions. For instance, the software:

- Considers that global variables and inputs of uncalled functions and methods are full range.
- Generates a `main` that calls uncalled functions in arbitrary order.

In addition, if you do not define a function but declare and call it in your code, Polyspace stubs the function. For a detailed list of assumptions, see “Code Prover Analysis Assumptions”.

You can use analysis options on the **Configuration** pane to change the default behavior and provide more context about your code. Performing contextual verification can result in more proven code and therefore fewer orange checks.

Control Variable Range

Use the following options. In the user interface of the Polyspace desktop products, the options appear under the **Code Prover Verification** node.

Option	Purpose
`Variables to initialize (-main-generator-writes-variables)`	Specify the global variables that Polyspace must consider as initialized despite no explicit initialization in the code.
`Constraint setup (-data-range-specifications)`	Specify range for global variables.

Control Function Call Sequence

1. Use the following options to call class methods. In the user interface of the Polyspace desktop products, the options appear under the **Code Prover Verification** node.

Option	Purpose
`Class (-class-analyzer)`	Specify classes whose methods the generated `main` must call.
`Functions to call within the specified classes (-class-analyzer-calls)`	Specify methods that the generated `main` must call.
`Analyze class contents only (-class-only)`	Specify that the generated `main` must call class methods only.
`Skip member initialization check (-no-constructors-init-check)`	Specify that the generated `main` must not check whether each class constructor initializes all class members.

2. Use the following options to call functions that are not class methods. In the user interface of the Polyspace desktop products, the options appear under the **Code Prover Verification** node.

Option	Purpose
`Initialization functions (-functions-called-before-main)`	Specify the functions that the generated `main` must call first.
`Functions to call (-main-generator-calls)`	Specify the functions that the generated `main` must call later.

See Also

More About

- “Verify C++ Classes” on page 18-9

Verify C Application Without main Function

Polyspace verification requires that your code must have a `main` function. You can do one of the following:

- Provide a `main` function in your code.
- Specify that Polyspace must generate a `main`.

Generate main Function

Before verification, specify one of the following options. In the user interface of the Polyspace desktop products, the options appear under the **Code Prover Verification** node.

Option	Description
`Verify whole application`	The verification stops if the software does not detect a `main`.
`Verify module or library (-main-generator)`	Before verification, Polyspace checks if your code contains a `main` function. If a `main` function exists, the software uses that `main`. Otherwise, the software generates a `main` using the options that you specify: • `Variables to initialize (-main-generator-writes-variables)` • `Initialization functions (-functions-called-before-main)` • `Functions to call (-main-generator-calls)`

Manually Write main Function

During automatic `main` generation, the software makes certain assumptions about the function call sequence or behavior of global variables. For instance, the default automatically generated `main` models the following behavior:

- The functions that you specify using the option `Functions to call (-main-generator-calls)` can be called in arbitrary order.
- In the beginning of each function body, global variables can have the full range of values allowed by their type.

To provide a more accurate model of the call sequence, you can manually write a `main` function for the purposes of verification. You can add this `main` function in a separate file to your project. In some cases, providing an accurate call sequence can reduce the number of orange checks. For example, in the following code, Polyspace assumes that `f` and `g` can be called in any order. Therefore, it produces an orange overflow for the case when `f` is called before `g`. If you know that `f` is called after `g`, you can write a `main` function to model this sequence.

```
static char x;
static int y;
```

```
void f(void)
{
    y = 300;
}

void g(void)
{
    x = y;
}
```

Example 1: main Calls One Function Before Another

Suppose you want to verify two functions func1 and func2 that have the following prototypes.

```
int func1(void *ptr, int x);
void func2(int x, int y);
```

You have the requirement that func1 is always called before func2.

To manually define a main that models this behavior:

1. Write a main containing declarations of a volatile variable for each function parameter type.
2. Write a loop with a volatile termination condition.

   The verification assumes that a volatile variable can have any value allowed by its type. Because the loop potentially terminates after any run, this condition models the fact that you call func1 and func2 an arbitrary number of times.
3. Inside this loop, call func2 after func1.

You can write the following main:

```
void main()
{
    volatile int random=0;
    volatile void * volatile ptr;
    while(random)
    {
        random = func1(ptr, random);
        func2(random, random);
    }
}
```

Example 2: main Calls One Function 10 Times Before Another

Suppose you want to verify two functions func1 and func2 with the following prototypes:

```
void func1(int);
void func2(void);
```

You know that when both func1 and func2 are called, func1 is always called 10 times before func2.

To manually define a main that models this behavior:

1. Write a main containing declarations of a volatile variable for each function parameter type.
2. In your main function, call func1 in a loop 10 times before func2.

For instance, you can write the following main:

```
void main(void) {
    int i=0;
    volatile int random=0;

    while (++i <= 10)
        func1(random);

    func2();

}
```

See Also

More About

- “Provide Context for C Code Verification” on page 18-2

Verify C++ Classes

In this section...
“Verification of Classes” on page 18-9
“Methods and Class Specifics” on page 18-11

Verification of Classes

Object-oriented languages such as C++ are designed for reusability. When developing code in such a language, you do not necessarily know every contexts in which the class is deployed. A class or a class family is safe for reuse if it free of defects for all possible contexts.

To make your classes safe against all possible contexts, perform a robustness verification and remove as many run-time errors as possible.

Polyspace Code Prover performs a robustness verification by default. If you provide the software the class definition together with the definition of the class methods, the software simulates all uses of the class. If some of the method definitions are missing, the software automatically stubs them.

1. The software verifies each constructor by creating an object using the constructor. If a constructor does not exist, the software uses the default constructor.
2. The software verifies the public, static and protected class methods of those objects assuming that:

   - The methods can be called in arbitrary order.
   - The method parameters can have any value in the range allowed by their data type.

   To perform this verification, by default, it generates a `main` function that calls the methods that are not called elsewhere in the code. If you want all your methods to be verified for all contexts, modify this behavior so that the generated `main` calls all public and protected methods instead of just the uncalled ones. For more information, see `Functions to call within the specified classes (-class-analyzer-calls)`.
3. The software calls the destructor of those objects (if they exist) and verifies them.

When verifying classes, Polyspace makes certain assumptions.

Code Construct	Assumption
Global variable	Unless explicitly initialized, in each method, global variables can have any value allowed by their type. For instance, in the following code, Polyspace assumes that `globvar1` can have any value allowed by its type. Therefore, an orange **Division by zero** appears on the division by `globvar1`. However, because `globvar2` is explicitly initialized, the **Division by zero** check on division by `globvar2` is green.

```
extern int fround(float fx);

// global variables
int globvar1;
int globvar2 = 100;

class Location
{
private:
    int x;

public:
    Location(int intx = 0) {
        x = intx;
    };

    void setx(int intx) {
        x = intx;
    };

    void fsetx(float fx) {
        int tx = fround(fx);
        if (tx / globvar1 != 0)
        {
            tx = tx / globvar2;
            setx(tx);
        }
    };
};
```

<table>
<tr><th>Code Construct</th><th>Assumption</th></tr>
<tr><td>Classes with undefined constructors</td><td>The members of the classes can be non-initialized.

In the following example, Polyspace assumes that <code>m_loc.x</code> can be non-initialized. Therefore, an orange Non-initialized variable error appears on <code>x</code> in the <code>getMember</code> method. Following the check, Polyspace assumes that the variable can have any value allowed by its type. Therefore, an orange Overflow appears on the addition operation in the <code>show</code> method.

<pre>class OtherClass
{
protected:
    int x;
public:
    OtherClass (int intx);
    int getMember(void) {
        return x;
    };
};

class MyClass
{
    OtherClass m_loc;
public:
    MyClass(int intx) : m_loc(0) {};
    void show(void) {
        int wx, wl;
        wx = m_loc.getMember();
        wl = wx + 2;
    };
};</pre></td></tr>
</table>

Methods and Class Specifics

- “Simple Class” on page 18-11
- “Template Classes” on page 18-13
- “Abstract Classes” on page 18-13
- “Static Classes” on page 18-14
- “Inherited Classes” on page 18-14
- “Simple Inheritance” on page 18-15
- “Multiple Inheritance” on page 18-16
- “Virtual Inheritance” on page 18-17
- “Class Integration” on page 18-17

Simple Class

Consider the following class:

`Stack.h`

```
#define MAXARRAY 100
```

```
class stack
{
  int array[MAXARRAY];
  long toparray;

public:
  int top (void);
  bool isempty (void);
  bool push (int newval);
  void pop (void);
  stack ();
};
```

stack.cpp

```
1 #include "stack.h"
2
3 stack::stack ()
4 {
5     toparray = -1;
6     for (int i = 0 ; i < MAXARRAY; i++)
7     array[i] = 0;
8 }
9
10 int stack::top (void)
11 {
12     int i = toparray;
13     return (array[i]);
14 }
15
16 bool stack::isempty (void)
17 {
18     if (toparray >= 0)
19         return false;
20     else
21         return true;
22 }
23
24 bool stack::push (int newvalue)
25 {
26     if (toparray < MAXARRAY)
27     {
28         array[++toparray] = newvalue;
29         return true;
30     }
31
32     return false;
33 }
34
35 void stack::pop (void)
36 {
37     if (toparray >= 0)
38         toparray--;
39 }
```

The class analyzer calls the constructor and then the methods in any order many times.

The verification of this class highlights two problems:

- The `stack::push` method may write after the last element of the array, resulting in the OBAI orange check at line 28.
- If called before `push`, the `stack::top` method will access element -1, resulting in the OBAI and NIV checks at line 13.

Fixing these problems will eliminate run-time errors in this class.

Template Classes

A template class allows you to create a class without explicit knowledge of the data type that the class operations handle. Polyspace cannot verify a template class directly. The software can only verify a specific instance of the template class. To verify a template class:

1. Create an explicit instance of the class.
2. Define a `typedef` of the instance and provide that `typedef` for verification.

In the following example, `calc` is a template class that can handle any data type through the identifier `myType`.

```
template <class myType> class calc
{
public:
    myType multiply(myType x, myType y);
    myType add(myType x, myType y);
};
template <class myType> myType calc<myType>::multiply(myType x,myType y)
{
    return x*y;
}
template <class myType> myType calc<myType>::add(myType x, myType y)
{
    return x+y;
}
```

To verify this class:

1. Add the following code to your Polyspace project.

   ```
   template class calc<int>;
   typedef calc<int> my_template;
   ```
2. Provide `my_template` as argument of the option **Class**. See `Class (-class-analyzer)`.

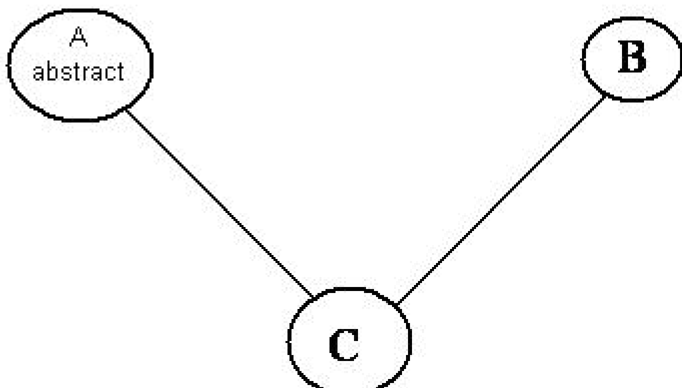
Abstract Classes

In the real world, an instance of an abstract class cannot be created, so it cannot be analyzed. However, it is easy to establish a verification by removing the pure declarations. For example, this can be accomplished via an abstract class definition change:

```
void abstract_func () = 0; by void abstract_func ();
```

If an abstract class is provided for verification, the software will make the change automatically and the virtual pure function (`abstract_func` in the example above) will then be ignored during the verification of the abstract class.

This means that no call will be made from the generated main, so the function is completely ignored. Moreover, if the function is called by another one, the pure virtual function will be stubbed and an orange check will be placed on the call with the message “call of virtual function [f] may be pure.”



Consider the following classes:

A is an abstract class

B is a simple class.

A and B are base classes of C.

C is not an abstract class.

As it is not possible to create an object of class A, this class cannot be analyzed separately from other classes. Therefore, you are not allowed to specify class A to the Polyspace class analyzer. Of course, class C can be analyzed in the same way as in the previous section “Multiple Inheritance.”

Static Classes

If a class defines a static methods, it is called in the generated main as a classical one.

Inherited Classes

When a function is not defined in a derived class, even if it is visible because it is inherited from a father's class, it is not called in the generated main. In the example below, the class `Point` is derived from the class `Location`:

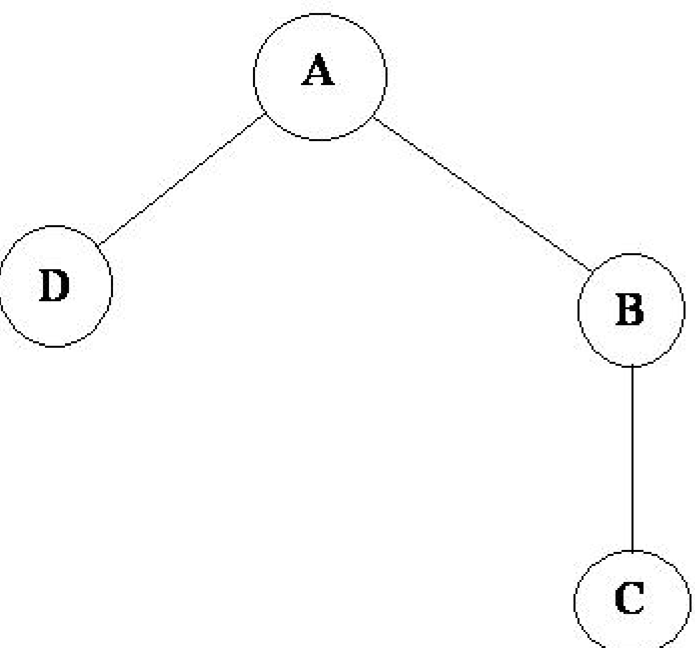
```
class Location
{
protected:
    int x;
    int y;
    Location (int intx, int inty);
public:
    int getx(void) {return x;};
    int gety(void) {return y;};
};
class Point : public Location
{
protected:
    bool visible;
```

```
public :
    Point(int intx, int inty) : Location (intx, inty)
    {
    visible = false;
    };
    void show(void) { visible = true;};
    void hide(void) { visible = false;};
    bool isvisible(void) {return visible;};
};
```

Although the two methods `Location::getx` and `Location::gety` are visible for derived classes, the generated main does not include these methods when analyzing the class `Point`.

Inherited members are considered to be volatile if they are not explicitly initialized in the father's constructors. In the example above, the two members `Location::x` and `Location::y` will be considered volatile. If we analyze the above example in its current state, the method `Location:: Location(constructor)` will be stubbed.

Simple Inheritance



Consider the following classes:

`A` is the base class of `B` and `D`.

`B` is the base class of `C`.

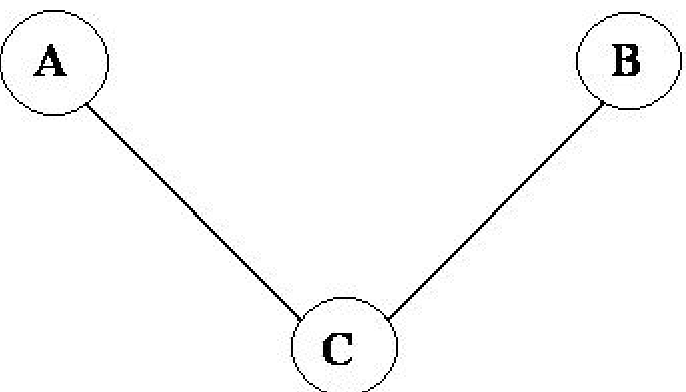
In a case such a this, Polyspace software allows you to run the following verifications:

1. You can analyze class `A` just by providing its code to the software. This corresponds to the previous “Simple Class” section in this chapter.
2. You can analyze class `B` class by providing its code and the class `A` declaration. In this case, `A` code will be stubbed automatically by the software.

3. You can analyze class B class by providing B and A codes (declaration and definition). This is a "first level of integration" verification. The class analyzer will not call A methods. In this case, the objective is to find bugs only in the class B code.
4. You can analyze class C by providing the C code, the B class declaration and the A class declaration. In this case, A and B codes will be stubbed automatically.
5. You can analyze class C by providing the A, B and C code for an integration verification. The class analyzer will call all the C methods but not inherited methods from B and A. The objective is to find only defects in class C.

In these cases, there is no need to provide D class code for analyzing A, B and C classes as long as they do not use the class (e.g., member type) or need it (e.g., inherit).

Multiple Inheritance



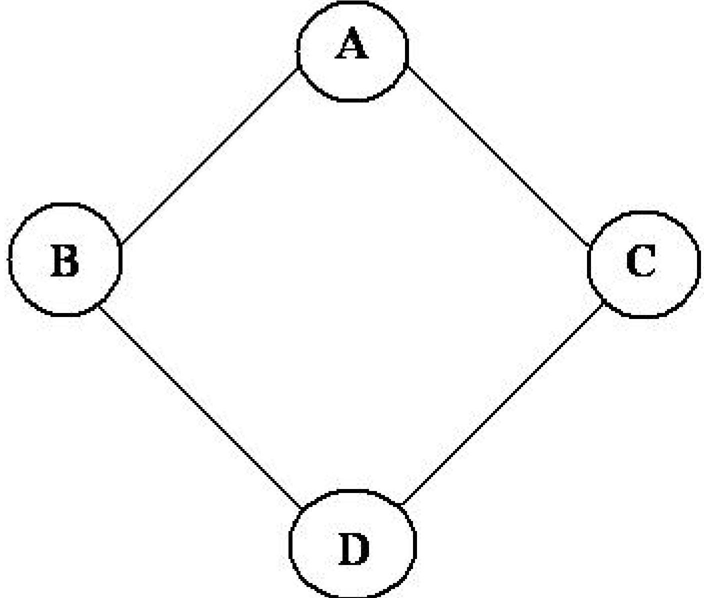
Consider the following classes:

A and B are base classes of C.

In this case, Polyspace software allows you to run the following verifications:

1. You can analyze classes A and B separately just by providing their codes to the software. This corresponds to the previous "Simple Class" section in this chapter.
2. You can analyze class C by providing its code with A and B declarations. A and B methods will be stubbed automatically.
3. You can analyze class C by providing A, B and C codes for an integration verification. The class analyzer will call all the C methods but not inherited methods from A and B. The objective is to find bugs only in class C.

Virtual Inheritance



Consider the following classes:

B and C classes virtually inherit the A class

B and C are base classes of D.

A, B, C and D can be analyzed in the same way as described in the previous section "Abstract Classes."

Virtual inheritance has no impact on the way of using the class analyzer.

Class Integration

Consider a C class that inherits from A and B classes and has object members of AA and BB classes.

A class integration verification consists of verifying class C and providing the codes for A, B, AA and BB. If some definitions are missing, the software will automatically stub them.

See Also

"Provide Context for C++ Code Verification" on page 18-4

Constrain Variable Ranges for Polyspace Analysis Using Manual Stubs and Manual main() Function

This topic applies primarily to a Polyspace Code Prover analysis.

To get the most precise results from a Polyspace Code Prover analysis, the program analyzed must be as complete as possible. However, for multiple reasons, you might not be able to provide a complete program for analysis. Even for complete applications, some information such as constraints on external inputs resides outside the program. This topic shows how to constrain your analysis in a way that allows you to get precise results even for incomplete applications.

For simpler constraints, Code Prover provides a graphical interface for constraint specification. See, for instance, "Specify External Constraints for Polyspace Analysis" on page 14-2. If the options in the interface are insufficient for your context and you need to fine-tune the analysis even further, you can explicitly define the program boundary in full detail. This topic provides a conceptual framework to get you started on defining the program boundary.

Code Prover Assumptions About Variable Ranges at Program Boundary

A Polyspace Code Prover analysis has to make assumptions about variable ranges at the program boundary in order to provide sound results.

When Polyspace Code Prover analyzes a program, the analysis checks all operations likely to cause certain types of run-time errors[3]. The objective is to mathematically prove that the operations checked do not cause run-time errors or definitely cause run-time errors. If the analysis fails to prove the presence or absence of an error, it highlights the operation in orange for manual inspection. A subset of these orange checks indicate possible run-time errors but depending on how you have set up the analysis, most of the orange checks might indicate that the analysis has insufficient knowledge about your application. See also "Orange Checks in Polyspace Code Prover" on page 33-2.

To get the most number of proven results from a Polyspace Code Prover analysis, the program analyzed must be as complete as possible. However, for multiple reasons, you might not always be able to provide a complete application for analysis. For instance, you might be analyzing libraries for robustness or your complete application might be too large for a Code Prover analysis to scale. In these cases, the program module that you analyze might be incomplete in two ways:

- There is no `main()` function that defines the entry point to your application.
- Not all function implementations are available to the analysis. For instance, the program module that you are analyzing might be calling functions defined in another module that is not provided to the analysis.

Consider this simple program module:

```
#include <stdint.h>
int8_t getAnInput(void);

int32_t calculateSum(int32_t numberToProcess) {
```

3 A Code Prover analysis checks all operations for run-time errors, but subject to verification assumptions. In particular, if the same value causes run-time errors in successive operations on an execution path, to avoid duplication of review efforts, only the first error is shown. See also "Code Prover Analysis Following Red and Orange Checks" on page 32-10.

```
    int32_t sum = 0;
    for(int32_t i = 0; i <= numberToProcess; i++) {
       sum += getAnInput();
    }
    return sum;
}
```

If you analyze this module with Code Prover, the following information is not available to the analysis:

- Implementation of the `getAnInput()` function.
- Call contexts of the `calculateSum()` function.

If the analyzed program is incomplete, Code Prover makes certain assumptions about the `main()` and undefined functions in order to continue with the analysis. The assumptions are broad enough to provide sound results but might be too broad for your context. For instance, if a function is not defined, Code Prover assumes a range of possible return values based on the function return type. In the preceding example, Code Prover assumes a range of [-128, 127] for the return value of `getAnInput()` based on the `int8_t` data type. However, this assumption leads to a wide range of values and the function might in reality be returning values in a narrower range.

Define Manual main() and Stubs to Constrain Ranges at Program Boundary

You can typically constrain a program boundary using Code Prover analysis options. For instance, if you want to constrain the input to `calculateSum()` or the return value of `getInput()`, you can use the option `Constraint setup (-data-range-specifications)`. However, the option supports specific types of constraints. To specify more complicated constraints, you have to define your own stub and `main()`.

Suppose that you know the following information:

- `calculateSum()` is always called with a positive argument that cannot exceed 100.
- `getAnInput()` returns a value that is either 0 or lies between 10 and 30 (inclusive of 10 and 30).

You can specify the following constrained ranges:

- Range [1,100] on the input to `calculateSum()`.

  To specify this range, you can define a manual `main()` that calls `calculateSum()`.
- Range 0 ∪ [10, 30] on the return value of `getAnInput()`.

  To specify this range, you can define a manual stub for the `getAnInput()` function.

You can define the manual `main()` and stub in files separate from the original program module and provide those files to the Code Prover analysis.

Define Manual main()

To impose a constraint on inputs to the `calculateSum()` function, define a `main()` function that calls `calculateSum()` with constrained arguments. Your `main()` function can simply be the following:

```
#include <stdint.h>
```

```
int32_t calculateSum(int32_t numberToProcess);
extern int32_t arg_calculateSum;

void main() {
   int32_t sum;
   unchecked_assert(arg_calculateSum >= 1 && arg_calculateSum <= 100);
   sum = calculateSum(arg_calculateSum);
}
```

Following the `unchecked_assert` macro, the verification assumes that the assertion is true for the remaining code within the block. Therefore, in this example, the verification assumes a constrained range [1,100] for the input to `calculateSum()`.

Define Manual Stub

To impose a constraint on the return value of `getAnInput()`, define a stub that constrains the range of an external variable before returning it from the function. The stub can look as follows:

```
#include <stdint.h>

extern int8_t arg_getAnInput;

int8_t getAnInput(void) {
   unchecked_assert(arg_getAnInput == 0 || (arg_getAnInput >= 10 && arg_getAnInput <= 30));
   return arg_getAnInput;
}
```

To check that the constraints are actually applied, verify the program module without and with the manual `main()` and stub. In the former case, you see two orange overflows that turn green when the constraints are applied. You can also verify the application of constraints using source code tooltips. For more information on the tooltips, see “Variable Ranges in Source Code Tooltips After Code Prover Analysis” on page 32-17.

See Also

`Constraint setup (-data-range-specifications)`

More About

- “Specify External Constraints for Polyspace Analysis” on page 14-2

Configure Code Prover Run-Time Checks

- "Modify or Disable Code Prover Run-Time Checks" on page 19-2
- "Modify Code Prover Run-Time Checks Through Code Behavior Specifications" on page 19-5

Modify or Disable Code Prover Run-Time Checks

A Code Prover analysis exhaustively checks source code for common run-time errors. The exhaustive nature of the static analysis is designed to prevent the errors from occurring at run time in safety critical software. Because of the safety critical intent of the analysis, Code Prover does not allow you to:

- Selectively disable specific run-time checks.
- Hide results of run-time checks from the results list through source code annotations. You can justify the results through annotations but the justified results continue to appear in the results list.

You can choose to suppress specific results by applying filters *after the analysis* or even creating filtered reports. If you create a filtered report from the Code Prover results, the report shows the filters and reflects your choices. For more information, see:

- “Filter and Group Results in Polyspace Desktop User Interface” on page 24-2 or “Filter and Sort Results in Polyspace Access Web Interface” on page 28-8
- “Generate Reports from Polyspace Results” on page 25-2 or `polyspace-report-generator`

However, you can modify the default behavior of certain checks and completely disable the checks for initialization. When you generate a report from the analysis results, the report shows the use of these options.

This topic lists the options that modify the default behavior of certain run-time checks. Note that though an option primarily addresses a specific type of check, checks of other types are also impacted. See “Code Prover Analysis Following Red and Orange Checks” on page 32-10.

Integer Overflow

Check	Default Behavior	Option
`Invalid shift operations`	Left shifts are not allowed on signed operands.	`Allow negative operand for left shifts (-allow-negative-operand-in-shift)`
`Overflow`	Signed integer overflows are forbidden.	`Overflow mode for signed integer (-signed-integer-overflows)`
`Overflow`	Unsigned integer overflows are not detected.	`Overflow mode for unsigned integer (-unsigned-integer-overflows)`

Floating Point Overflow

Check	Default Behavior	Option
• Overflow • Invalid operation on floats	Non-finite floats are not considered.	• Consider non finite floats (-allow-non-finite-floats) • Infinities (-check-infinite) • NaNs (-check-nan)
Subnormal float	Subnormal results are not detected.	Subnormal detection mode (-check-subnormal)

Initialization

Check	Default Behavior	Option
• Non-initialized local variable • Non-initialized variable • Non-initialized pointer • Return value not initialized	Checks for initialization are performed only when a variable is read.	Disable checks for non-initialization (-disable-initialization-checks)
Global variable not assigned a value in initialization code	Checks for global variable initialization is performed only when the variable is read.	• Ignore default initialization of global variables (-no-def-init-glob) • Check that global variables are initialized after warm reboot (-check-globals-init)

Library Functions

Check	Default Behavior	Option
Invalid use of standard library routine	Only Standard Library routines are checked for validity of arguments. User-defined library functions are not checked.	-code-behavior-specifications

Pointers

Check	Default Behavior	Option
Illegally dereferenced pointer	Pointers to a structure field cannot point to another field.	Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)
Illegally dereferenced pointer	Pointers to a structure must have enough buffer for the entire structure.	Allow incomplete or partial allocation of structures (-size-in-bytes)
Illegally dereferenced pointer	Stack pointer dereference outside scope is not detected.	Detect stack pointer dereference outside scope (-detect-pointer-escape)
Correctness condition	Function pointer mismatches are not allowed.	Permissive function pointer calls (-permissive-function-pointer)

Unreachable Code or Dead Code

Checker	Default Behavior	Option
• Function not called • Function not reachable	Uncalled functions and functions called from unreachable code are not reported.	Detect uncalled functions (-uncalled-function-checks)

See Also

More About

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

Modify Code Prover Run-Time Checks Through Code Behavior Specifications

Polyspace Code Prover exhaustively checks C/C++ code for run-time errors. For greater analysis precision, you can provide additional information outside your code.

You can specify most external information using the option `-code-behavior-specifications`. The option allows you to associate behaviors with elements in your code. For instance, you can map a custom library function to a standard function or specify that a function acts as a memory management function. The option takes a file in one of these formats as argument:

- XML
- Datalog

Only the XML format applies to Polyspace Code Prover. This topic describes the supported check modifications. For the full list of options that can modify run-time checks, see “Modify or Disable Code Prover Run-Time Checks” on page 19-2.

The code behavior specifications XML file associates specific behaviors with functions (or defines behaviors at global scope). The following sections list behaviors that are relevant to a Code Prover analysis. For behaviors relevant to Bug Finder, see “Modify Code Prover Run-Time Checks Through Code Behavior Specifications” on page 19-5.

You can also see the supported behaviors in the sample template file `code-behavior-specifications-template.xml` provided with a Polyspace installation. The file is in *`polyspaceroot`*`\polyspace\verifier\cxx\` where *`polyspaceroot`* is the Polyspace installation folder.

Mapping to Standard Function for Precision Improvement

XML Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
    <functions>
        <function name="custom_function" std="std_function">
        </function>
    </functions>
</specifications>
```

Use this entry in the XML file to reduce the number of orange checks from imprecise Code Prover analysis of your function (or false negatives from an imprecise Bug Finder analysis). Sometimes, the verification does not analyze certain kinds of functions precisely because of inherent limitations in static verification. In those cases, if you find a standard function that is a close analog of your function, use this mapping. Though your function itself is not analyzed, the analysis is more precise at the locations where you call the function. For instance, if the verification cannot analyze your function `cos32` precisely and considers full range for its return value, map it to the `cos` function for a return value in [-1,1].

The verification ignores the body of your function. However, the verification emulates your function behavior in the following ways:

- The verification assumes the same return values for your function as the standard function.

  For instance, if you map your function `cos32` to the standard function `cos`, the verification assumes that `cos32` returns values in [-1,1].
- The verification checks for the same issues as it checks with the standard function.

  For instance, if you map your function `acos32` to the standard function `acos`, the `Invalid use of standard library routine` check determines if the argument of `acos32` is in [-1,1].

The functions that you can map to include:

- Standard library functions from `math.h`.
- Memory management functions from `string.h`.
- `__ps_meminit`: A function specific to Polyspace that initializes a memory area.

  Sometimes, the verification does not recognize your memory initialization function and produces an orange `Non-initialized local variable` check on a variable that you initialized through this function. If you know that your memory initialization function initializes the variable through its address, map your function to `__ps_meminit`. The check turns green.
- `__ps_lookup_table_clip`: A function specific to Polyspace that returns a value within the range of the input array.

  Sometimes, the verification considers full range for the return values of functions that look up values in large arrays (look-up table functions). If you know that the return value of a look-up table function must be within the range of values in its input array, map the function to `__ps_lookup_table_clip`.

  In code generated from models, the verification by default makes this assumption for look-up table functions. To identify if the look-up table uses linear interpolation and no extrapolation, the verification uses the function names. Use the mapping only for handwritten functions, for instance, functions in a C/C++ S-Function block. The names of those functions do not follow specific conventions. You must explicitly specify them.

Mapping to Standard Function for Concurrency Detection

XML Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
    <functions>
        <function name="custom_function" std="std_function">
        </function>
    </functions>
</specifications>
```

Use this entry in the XML file for automatic detection of thread-creation functions and functions that begin and end critical sections. Polyspace supports automatic detection for certain families of multitasking primitives only. Extend the support using this XML entry.

If your thread-creation function, for instance, does not belong to one of the supported families, map your function to a supported concurrency primitive.

Extending Initialization Checks

XML Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
    <functions>
        <function name="my_func">
            <check name=name="ARGUMENT_POINTS_TO_INITIALIZED_VALUE" arg="n"/>
        </function>
    </functions>
</specifications>
```

Use this entry in the XML file to specify if the pointer argument to a function *my_func* must point to an initialized buffer. The number *n* specifies which argument must be checked for buffer initialization.

Specifying Contribution from Stubbed Functions to Stack Size

XML Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
    <functions>
        <function name="my_func">
            <behavior name="STACK_SIZE_LOCAL_VARIABLES" value="size"/>
        </function>
    </functions>
</specifications>
```

Use this entry in the XML file to specify that local variables in a function *my_func* contribute *size* bytes to the stack size. Stack size computation for callers of *my_func* take this contribution into account.

Instead of a fixed size, you can also specify a lower and upper bound on the stack size contribution from a function. Enter a minimum local variable size *min_size* and a maximum size *max_size* for the `value` attribute as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
    <functions>
        <function name="my_func">
            <behavior name="STACK_SIZE_LOCAL_VARIABLES" value="min_size,max_size"/>
        </function>
    </functions>
</specifications>
```

Note that for template functions, you have to specify the same stack size contribution from all instantiations of the functions. You cannot specify separate values for different instantiations.

See Also

`-code-behavior-specifications`

Related Examples

- "Modify Default Behavior of Bug Finder Checkers"

Configure Comment Import from Previous Results

- "Import Review Information from Previous Polyspace Analysis" on page 20-2
- "Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results" on page 20-5

Import Review Information from Previous Polyspace Analysis

This topic describes how to import review information from previous results that are not already uploaded to Polyspace Access. For information on importing from results uploaded to Polyspace Access, see “Import Review Information from Existing Polyspace Access Projects”.

After you have reviewed analysis results, you can reuse information from the review for subsequent analyses. If you specify a result status or severity or add notes in your results file, they carry over to the results of the next analysis on the same project. If you add the same information as comments to your code (annotate), they carry over to any subsequent analysis of the code, whether in the same project or not. You can also hide results using code annotations. For more information on commenting, see “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2.

This topic shows how to import review information from one result file to another. Importing the review information saves you from reviewing already justified results. For instance, after you import the information, on the **Results List** pane (user interface of desktop products), clicking the icon skips justified results. Using this icon, you can browse through unreviewed results. You can also filter the justified checks from display.

Automatic Import from Last Analysis

By default, in the Polyspace user interface (desktop products only), review information is imported automatically from the most recent analysis on the project module. You can disable this default behavior.

1 Select **Tools** > **Preferences**, which opens the Polyspace Preferences dialog box.
2 Select the **Project and Results Folder** tab.
3 Under **Import Comments**, clear **Automatically import comments from last verification**.
4 Click **OK**.

If you run analysis at the command line (and do not upload results to the Polyspace Access web interface), you have to explicitly import from another set of results. See “Command Line”.

Import from Another Analysis Result

You can import review information directly from another Polyspace result to the current result.

If a result is found in both a Bug Finder and Code Prover analysis, you can add review information to the Bug Finder result and import to the Code Prover result. For instance, most coding rule checkers are common to Bug Finder and Code Prover. You can add review information to coding rule violations in Bug Finder and import to the same violations in Code Prover.

User Interface (Desktop Products Only)

To import review information from another set of results:

1 Open the current analysis results.
2 Select **Tools** > **Import Comments**.
3 Navigate to the folder containing your previous results.

4 Select the other results file (with extension `.psbf` or `.pscp`) and then click **Open**.

   The review information from the previous results are imported into the current results.

Command Line

Use the option `-import-comments` during analysis to import comments from a previous verification.

To import review information from multiple results, use the `polyspace-comments-import` command.

Import Algorithm

You can directly import review information from another set of results into the current results. However, it is possible that part of your review information is not imported to a subsequent analysis because:

- You have changed your source code so that the line with a previous result is not exactly identical to the line in the current run.

  The comment import tool accounts for additional code that simply shifts an existing line. For instance, the tool recognizes that line 10 in Run 1 and line 12 in Run 2 have the same statement. If a division by zero occurs on line 10 in Run 1 and you have not fixed the issue in Run 2, the result along with associated review information are imported to line 12 in Run 2.

  - Run 1:

    ```
    10 baseLine = min/numRecipients;
    11
    12
    ```

  - Run 2:

    ```
    10 /* Calculate a baseline per recipient
    11    based on minimum available resources */
    12 baseLine = min/numRecipients;
    ```

  However, if you change the line content itself, for instance, change `numRecipient` to `numReceiver`, the result and review information are not imported.

- You have changed your source code so that the Code Prover result color has changed.
- You entered new review information for the same result.

If the content of a line does not change and shows the same result as the previous analysis, the review information is imported. In unlikely scenarios, you might get the same result on the same line despite changing previous lines that lead to the result. Your review information from a previous analysis is then imported to the new result. If you justified the previous result with a status such as `Not a defect`, it is likely that you want to continue this justification with the new result. For instance, if you accepted an overflow previously because you accounted for a wrap-around behavior after the overflow, you are likely to accept the overflow whatever the cause. In a few cases, you might want to review the result again and might not be aware that the result merits another review. To avoid this situation:

- When justifying nonlocal results that are related to previous events, use careful judgement.

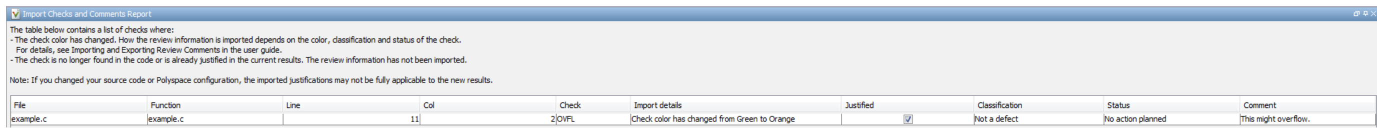
- For critical components, conduct periodic assessments of justified results to see if the justifications still apply. Such assessments are useful specially for the Code Prover run-time checks.

View Imported Review Information That Does Not Apply

In the Polyspace user interface (desktop products only), the Import Checks and Comments Report highlights differences between two analysis results. When you import review information from a previous analysis, you can see this report. If you have closed the report after an import, to review the report again:

1. Select **Window > Show/Hide View > Import Comments Report**.

   The Import Checks and Comments Report opens, highlighting differences in the two results.



2. Review the differences between the two results.

Your review information can differ between two results because of the following reasons:

- In Code Prover, if the check color changes, Polyspace imports the **Comment** field but not the **Status** field. In addition, Polyspace imports the **Severity** and **Justified** fields depending on the color change.

Color Change	Severity	Justified
Orange or red to green	Not imported	Imported
Gray to green	Not imported	Imported, if the **Severity** was set to `High`, `Medium` or `Low`.
Red to orange or vice versa	Imported	Imported
Green to red/orange/gray	Not imported	Not imported

- If a result no longer appears in the code, Polyspace highlights only the change in the Import Checks and Comments Report. It does not import review information from the previous result.
- If you have already entered different review information for the same check, Polyspace highlights only the change in the Import Checks and Comments Report. It does not import review information from the previous result.

See Also

`-import-comments` | `polyspace-comments-import`

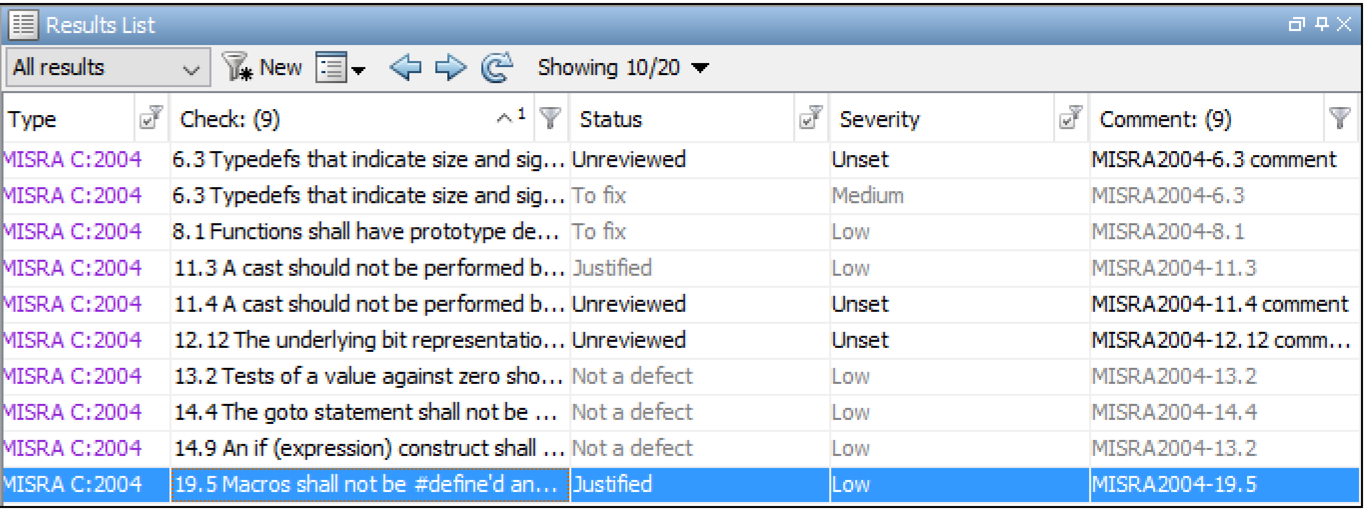
Related Examples

- “Import Review Information from Existing Polyspace Access Projects” on page 27-5

Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results

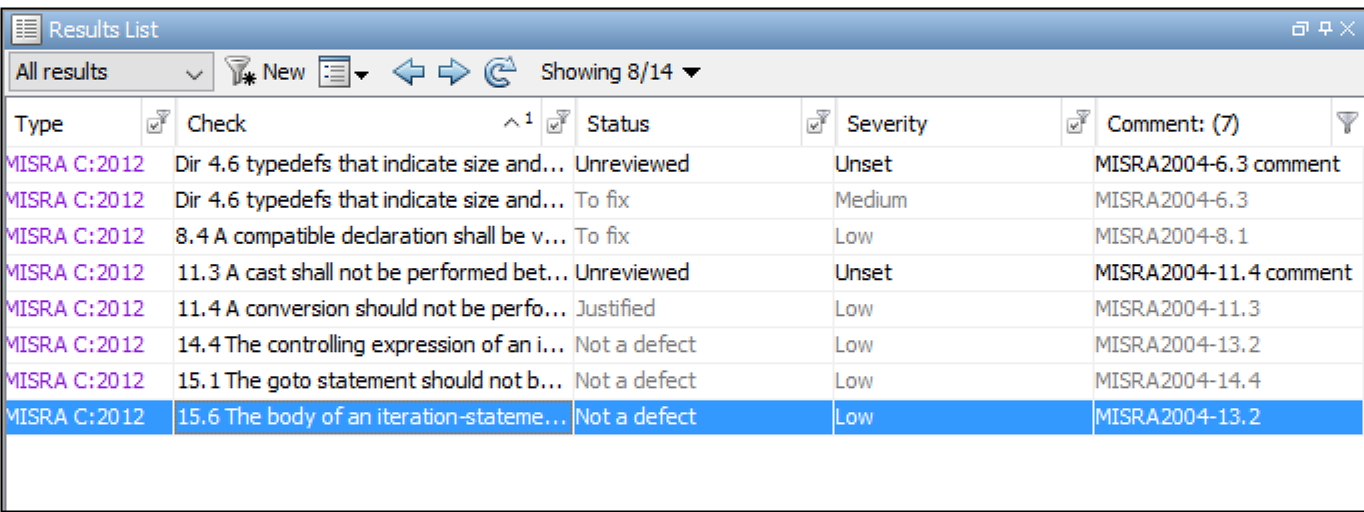
When you check your code for MISRA C: 2012 violations, Polyspace imports justifications of MISRA C: 2004 violations from previous analyses (if they exist). You can upgrade from checking of MISRA C: 2004 rules to MISRA C: 2012 rules while retaining your justifications. For general rules on comment import, see “Import Review Information from Previous Polyspace Analysis”.

The software maps MISRA C: 2004 **Status**, **Severity**, and **Comment** values that you added through the user interface or code annotations to the corresponding MISRA C: 2012 results, if the results exist. For more information about mapping, consult addendum one of the MISRA C: 2012 publication.



Type	Check: (9)	Status	Severity	Comment: (9)
MISRA C:2004	6.3 Typedefs that indicate size and sig…	Unreviewed	Unset	MISRA2004-6.3 comment
MISRA C:2004	6.3 Typedefs that indicate size and sig…	To fix	Medium	MISRA2004-6.3
MISRA C:2004	8.1 Functions shall have prototype de…	To fix	Low	MISRA2004-8.1
MISRA C:2004	11.3 A cast should not be performed b…	Justified	Low	MISRA2004-11.3
MISRA C:2004	11.4 A cast should not be performed b…	Unreviewed	Unset	MISRA2004-11.4 comment
MISRA C:2004	12.12 The underlying bit representatio…	Unreviewed	Unset	MISRA2004-12.12 comm…
MISRA C:2004	13.2 Tests of a value against zero sho…	Not a defect	Low	MISRA2004-13.2
MISRA C:2004	14.4 The goto statement shall not be …	Not a defect	Low	MISRA2004-14.4
MISRA C:2004	14.9 An if (expression) construct shall …	Not a defect	Low	MISRA2004-13.2
MISRA C:2004	19.5 Macros shall not be #define'd an…	Justified	Low	MISRA2004-19.5

If you are transitioning from MISRA C: 2004 to MISRA C: 2012, you do not have to review results that you have already justified.



Type	Check	Status	Severity	Comment: (7)
MISRA C:2012	Dir 4.6 typedefs that indicate size and…	Unreviewed	Unset	MISRA2004-6.3 comment
MISRA C:2012	Dir 4.6 typedefs that indicate size and…	To fix	Medium	MISRA2004-6.3
MISRA C:2012	8.4 A compatible declaration shall be v…	To fix	Low	MISRA2004-8.1
MISRA C:2012	11.3 A cast shall not be performed bet…	Unreviewed	Unset	MISRA2004-11.4 comment
MISRA C:2012	11.4 A conversion should not be perfo…	Justified	Low	MISRA2004-11.3
MISRA C:2012	14.4 The controlling expression of an i…	Not a defect	Low	MISRA2004-13.2
MISRA C:2012	15.1 The goto statement should not b…	Not a defect	Low	MISRA2004-14.4
MISRA C:2012	15.6 The body of an iteration-stateme…	Not a defect	Low	MISRA2004-13.2

Mapping Multiple MISRA C: 2004 Annotations to the Same MISRA C: 2012 Result

When you justify MISRA C: 2004 violations by using code block syntax or multiple line annotation syntax, and multiple violations map to the same MISRA C: 2012 rule, Polyspace does not import each result justification. Instead, the software imports only one set of **Status**, **Severity**, and **Comment** values and applies these values to all the instances of that particular MISRA C: 2012 rule violation.

For example, suppose that you analyze your code and find violations of MISRA C: 2004 rules 16.3 and 16.5. You can justify these results by using the annotation syntax where you enter a different status and explanatory comment for each rule.

```
//polyspace-begin misra2004:16.3 [Status 1] "Explanatory comment 1"
//polyspace-begin misra2004:16.5 [Status 2] "Explanatory comment 2"

code block start;
/* This block of code contains violations of
MISRA C:2004 rules 16.3 and 16.5 */
code block end;

//polyspace-end misra2004:16.3
//polyspace-end misra2004:16.5
```

The previous violations map to MISRA C: 2012 rule 8.2. When you check your annotated code against MISRA C: 2012 rules, Polyspace imports only the first line of annotations (for rule 16.3) and applies it to all rule 8.2 results. The second line of annotations for rule 16.5 is ignored. In the **Results List** pane, all violations of rule 8.2 have the **Status** column set to **`Status 1`** and the **Comment** column set to **`"Explanatory comment 1"`**.

Note The **Output Summary** pane displays a warning message for every result where the imported annotation conflicts with the original annotation. After you import your MISRA C: 2004 annotations, check that a justified status has not been assigned to results you intend to investigate or fix.

See Also

`Check MISRA C:2004 (-misra2)` | `Check MISRA C:2012 (-misra3)`

More About

- “Annotate Code and Hide Known or Acceptable Results” on page 30-2

Review Results in Polyspace User Interface

Interpret Polyspace Code Prover Results

- “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2
- “Dashboard in Polyspace Desktop User Interface” on page 21-7
- “Concurrency Modeling in Polyspace Desktop User Interface” on page 21-11
- “Results List in Polyspace Desktop User Interface” on page 21-12
- “Source Code in Polyspace Desktop User Interface” on page 21-15
- “Result Details in Polyspace Desktop User Interface” on page 21-21
- “Call Hierarchy in Polyspace Desktop User Interface” on page 21-24
- “Variable Access in Polyspace Desktop User Interface” on page 21-27
- “Understanding Changes in Polyspace Results After Product Upgrade” on page 21-33

Interpret Code Prover Results in Polyspace Desktop User Interface

This topic shows how to review Code Prover results in the user interface of the Polyspace desktop products. To see how to review results in the Polyspace Access web interface, see “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2.

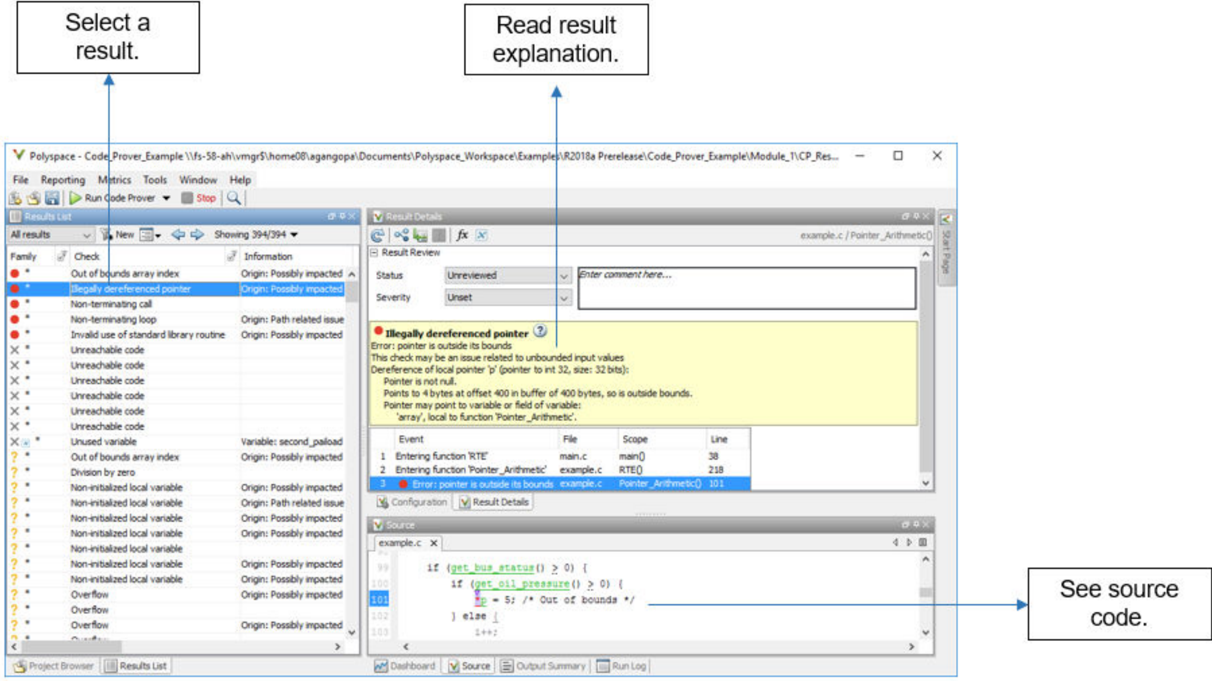
When you open the results of a Polyspace Code Prover analysis, you see a list on the **Results List** pane. The list consists of run-time checks, coding rule violations, code metrics and global variable usage.

You can first narrow down the focus of your review:

- Use filters on the results list columns. For instance, you can first focus on red checks.
- Organize results by file and function. Use the icon above the list.

  Because the results of a Code Prover run-time check are dependent on the results of previous checks, it helps to go through run-time checks from the beginning to the end of a function.

See also “Filter and Group Results in Polyspace Desktop User Interface” on page 24-2. 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

Interpret Message

The first step is to understand what the issue is. Read the message on the **Result Details** pane and the related line of code on the **Source** pane.

At this point, you might be ready to decide whether to fix the issue.

? Out of bounds array index
Warning: array index may be outside bounds : [0..126]
This check may be an issue related to unbounded input values
Local variable main.PORT_A that is declared volatile in main.c line 48 may lead to imprecise check
Local variable main.PORT_B that is declared volatile in main.c line 49 may lead to imprecise check
array size: 127
array index value: [0 .. 555]

The message consists of several parts:

- Check color and icon: See “Code Prover Result and Source Code Colors” on page 32-2. In case of checks for run-time errors:
  - ● : Red indicates a definite error.
  - ?: Orange indicates a possible error.
  - ✕: Gray indicates unreachable code.
  - ✓: Green indicates that a specific error cannot happen.
- Description of the run-time check.

  In the preceding example, the check determines if an array index goes outside the array bounds.
- Values relevant to the run-time check.

  In the example, the message states the array size (127), the array bounds (0..126), and the range of values that the array index variable can take at that point in the code (0..555).
- Relevant sources of imprecision (for orange checks).

  In the example, the message states that two volatile variables might be responsible for the check.

See Variable Ranges in Source Code Tooltips

On the **Source** pane, variables and operations with tooltips are underlined.

```
output_v1 = s8_ret + 15;
```

In this example, tooltips appear on:

- `s8_ret`: You see its data type and range of values before the + operation.

  If a data type conversion occurs during the + operation, you also see this conversion in the tooltip.
- +: You see the value of the left and right operand, and the result.

- =: You see any data type conversion that occurs during the assignment and the result.

Get Additional Help

Sometimes, you need additional help for certain results. To open a help page for the selected result, click the icon. See code examples that illustrate the result.

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 possibly a previous `if` or `while` condition that is always false.

Navigate in Source Code

Sometimes, the **Result Details** pane shows one sequence of events that leads to the result. However, in most situations, you have to find your own navigation pathways through the code. Using tooltips on variables, follow the propagation of variable ranges as you navigate through the code.

```
int func (int var) { /* Initial range of var */
     …
     var -= get ();  /* New range of var */
     …
     set(&var);     /* New range of var */
}
```

Use these quick navigation pathways in the user interface:

- Search for all references to a variable and browse through them.

  Right-click the variable name on the **Source** pane and select **Search For All References**. Alternatively, double-click the variable. These options perform more than a string match. The options show only instances of a specific variable and not other variables with the same name in other scopes.
- Navigate from a function call to its definition.

  Right-click the function name on the **Source** pane. Select **Go To Definition**.
- Navigate from a function to its callers and callees.

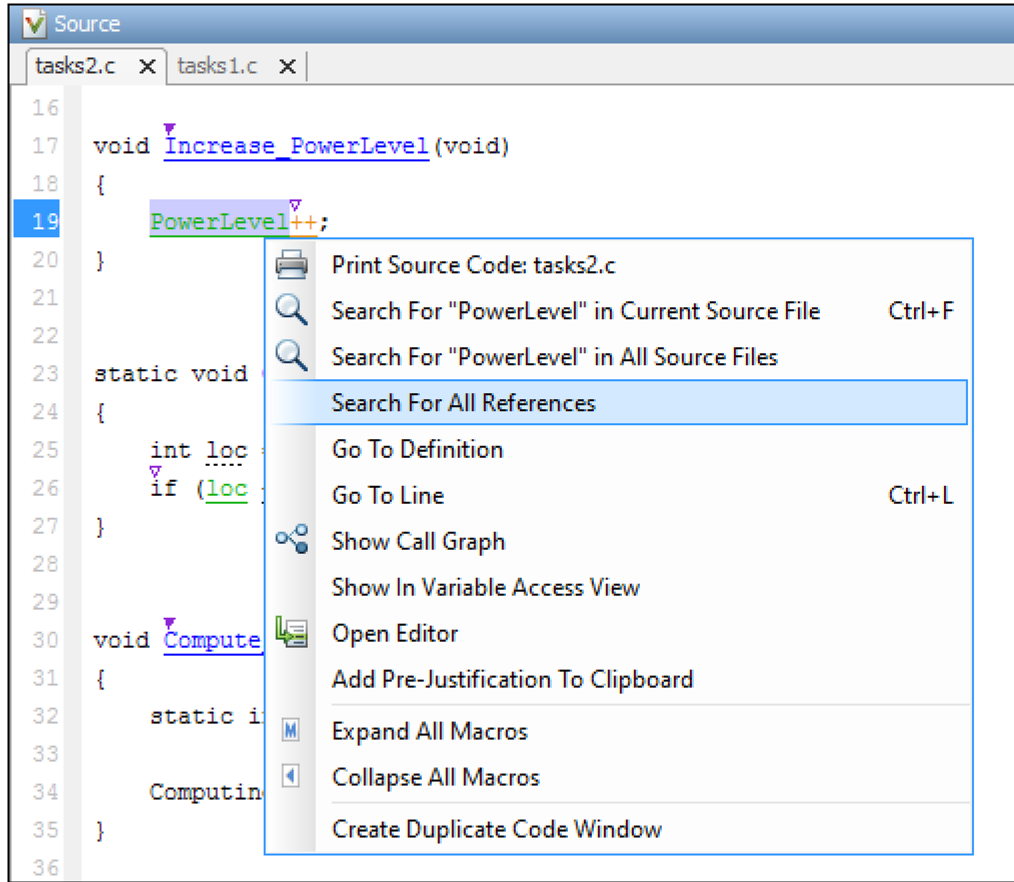
  Click the icon on the **Result Details** pane. You see the function containing the result, with its callers and callees. Click a caller or callee name to navigate to the call site. Double-click a name to navigate to the definition.

  Alternatively, click the icon to see a graphical representation of the call sequence leading to the result. To navigate to functions in this sequence, click through nodes in the graph.
- Navigate from a function call or loop keyword to an error in the function or loop body.

  If the error occurs only in a specific function call or specific loop iteration, the function call or loop iteration is highlighted red. Right-click the red function call or loop keyword. Select **Go To Cause** if the option is available.

- Navigate across all instances of a global variable.

  Click the icon on the **Result Details** pane. See all global variables in the result and read/write operations on them.



Before you begin navigating through pathways in your code, determine what you are looking for and choose the appropriate navigation tool. For instance:

- To investigate a **Non-initialized variable** check, you might want to make sure that the variable is not initialized at all. Look for previous instances of the variable and see if it is initialized.
- 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.

For other examples of what to look for, see “Reviewing Code Prover Run-Time Checks” on page 32-7. After you navigate away from the current result, use the icon on the **Result Details** pane to return to that result.

If you click a source code token containing a result, the previous result selection on the **Results List** and 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 in the **Source** pane and select **Create Duplicate Code Window**. Right-click 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 in the original file window. After the investigation is complete, close the duplicate window.

See Also

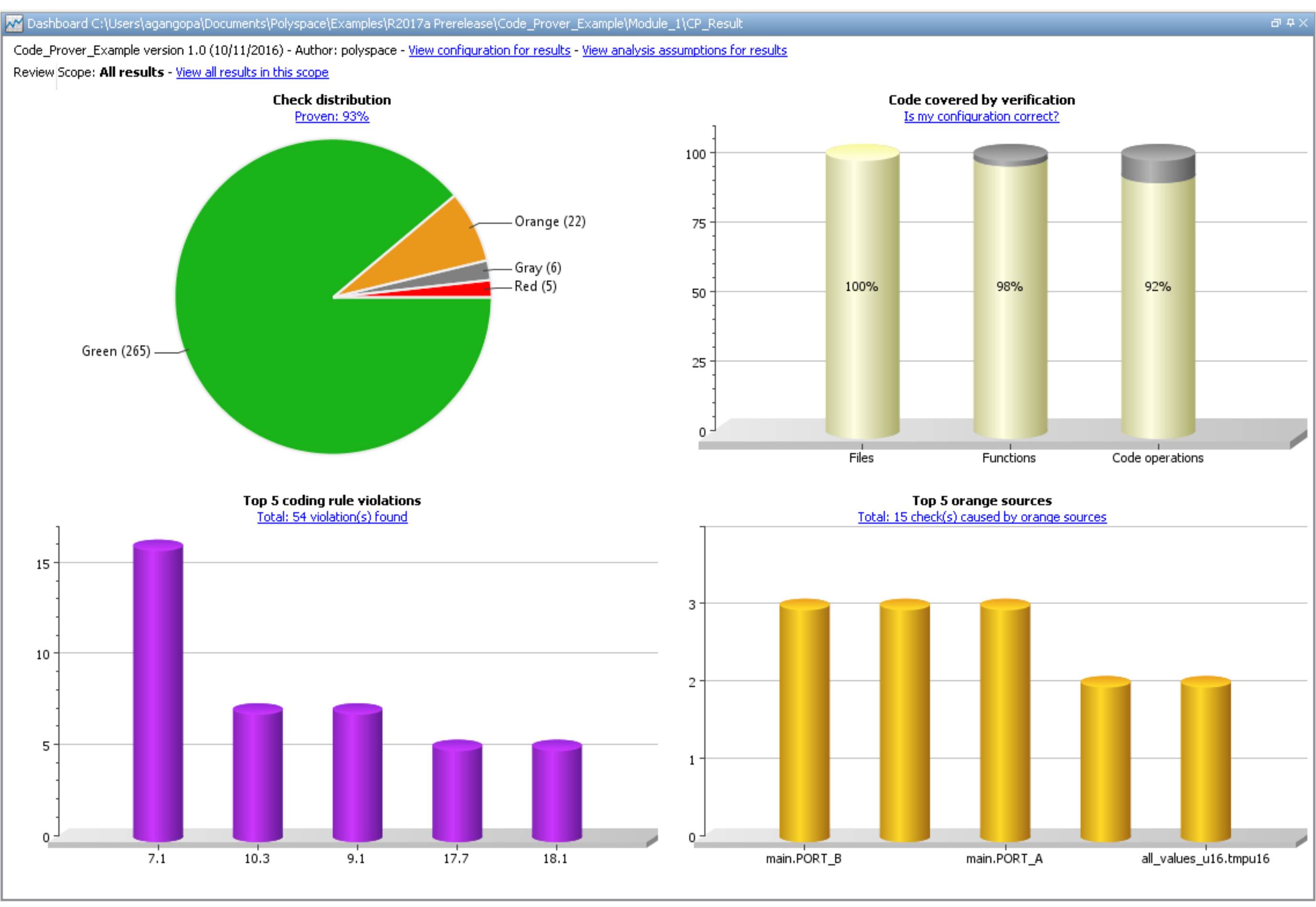
More About

- "Filter and Group Results in Polyspace Desktop User Interface" on page 24-2
- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" on page 23-2

Dashboard in Polyspace Desktop User Interface

This topic focuses on the Polyspace desktop user interface. To learn about the equivalent pane in the Polyspace Access web interface, see “Dashboard in Polyspace Access Web Interface” on page 26-7.

The **Dashboard** pane in the Polyspace user interface provides statistics on the verification results in a graphical format.



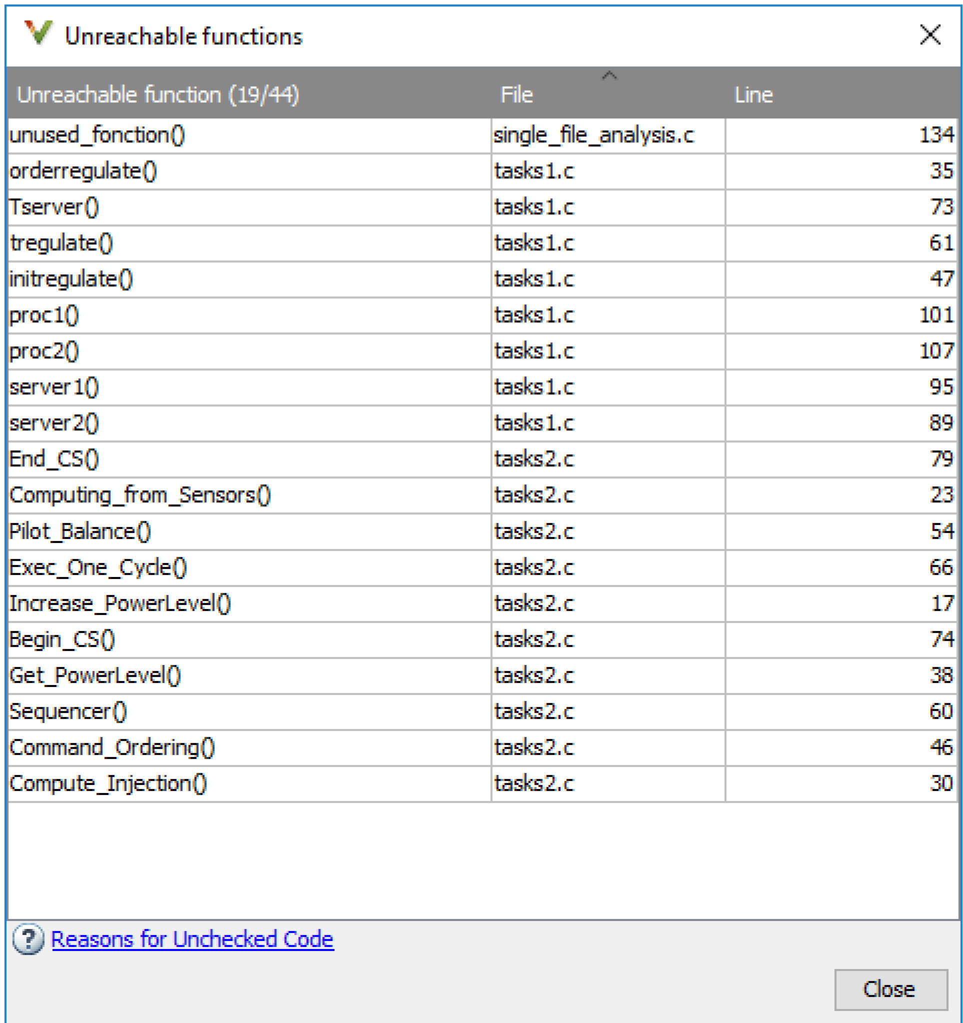
On this tab, you can view four graphs and charts.

Code Covered by Verification

This column graph displays:

- The percentage of files checked for run-time errors (verified). You can see this percentage in the **Files** column.
- The percentage of functions in verified files that are checked for run-time errors (verified). You can see this percentage in the **Functions** column.
- The percentage of elementary operations in verified functions that are checked for run-time errors. You can see this percentage in the **Code operations** column.

Click the column graph to open the **Unreachable functions** window.



This window contains:

- The number of functions that are unreachable in the format, `Number of unreachable functions/Total number of functions`.
- A list of unreachable functions along with the file and line number where they are defined. Selecting a function displays the function definition in the **Source** pane.

A low coverage can indicate an early red check or missing function call. Consider the following code:

```
void coverage_eg(void)
{
  int x;

  x = 1 / x;
  x = x + 1;
  propagate();
}
```

Verification generates only one red **Non-initialized local variable** check, for a read operation on the variable `x` — see line 5. The software does not display checks for these elementary operations:

- **Division by zero** check on the division.

- **Overflow** check on the division result.
- **Overflow** check on the addition result.
- **Non-initialized local variable** check when `x` is read in the operation `x=x+1`.

As the software displays only one out of the five operation checks for the code, the percentage of elementary operations covered is 1/5 or 20%. The software does not take into account the checks inside the unreachable function `propagate()`. For more information, see “Reasons for Unchecked Code” on page 34-77.

Check Distribution

This pie chart displays the number of checks of each color. For a description of the check colors, see “Code Prover Result and Source Code Colors” on page 32-2.

Using this pie chart, you can obtain an estimate of:

- The number of checks to review.
- The selectivity of your verification — the fraction of checks that are not orange.

  You can follow certain coding rules or specify certain verification options to reduce the number of orange checks. See “Reduce Orange Checks in Polyspace Code Prover” on page 33-17.

Top 5 Orange Sources

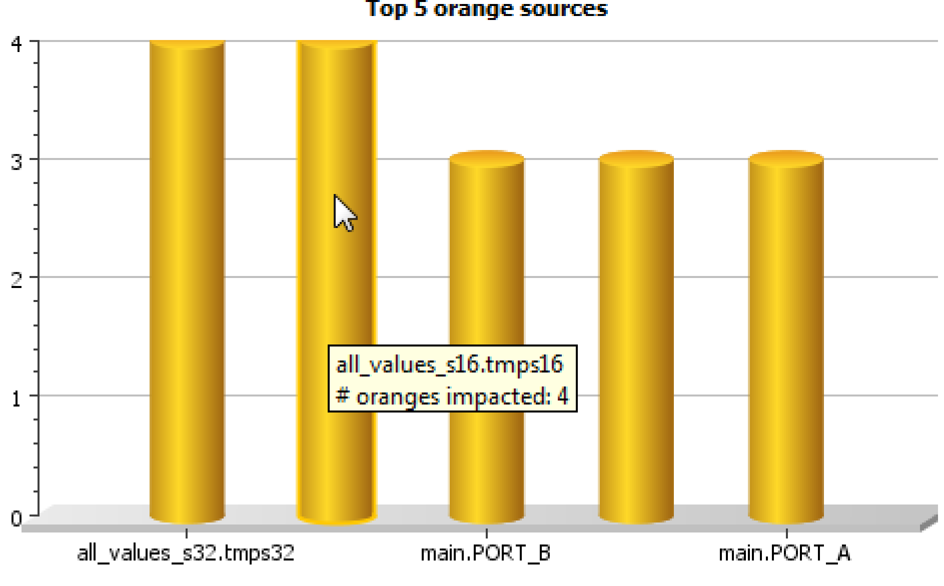
An orange source is a variable or function that leads to an orange check. This column graph displays five orange sources affecting the most number of checks.

An orange source can cause multiple orange checks in Code Prover. When you click an orange source in this graph, the **Results List** pane shows only the orange checks coming from this source.

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;
}
```

Each column represents an orange source. The columns are arranged in the order of number of checks affected. The height of the column indicates the number of checks affected by the corresponding orange source. Place your cursor on a column to open a tooltip showing the source name and the number of checks affected by the source.



Using this chart, you can:

- View the five sources affecting the most number of checks. Select a column to view further details of the corresponding orange source in the **Orange Sources** pane.
- Prioritize your review of orange checks. If there are sources affecting a large number of orange checks, address those sources if possible before you begin a systematic review of orange checks. See "Create Constraint Template from Code Prover Analysis Results" on page 14-3.

Top 5 Coding Rule Violations

This column graph displays the five most violated coding rules. Each column represents a coding rule and is indexed by the rule number. The height of the column indicates the number of violations of the coding rule represented by that column.

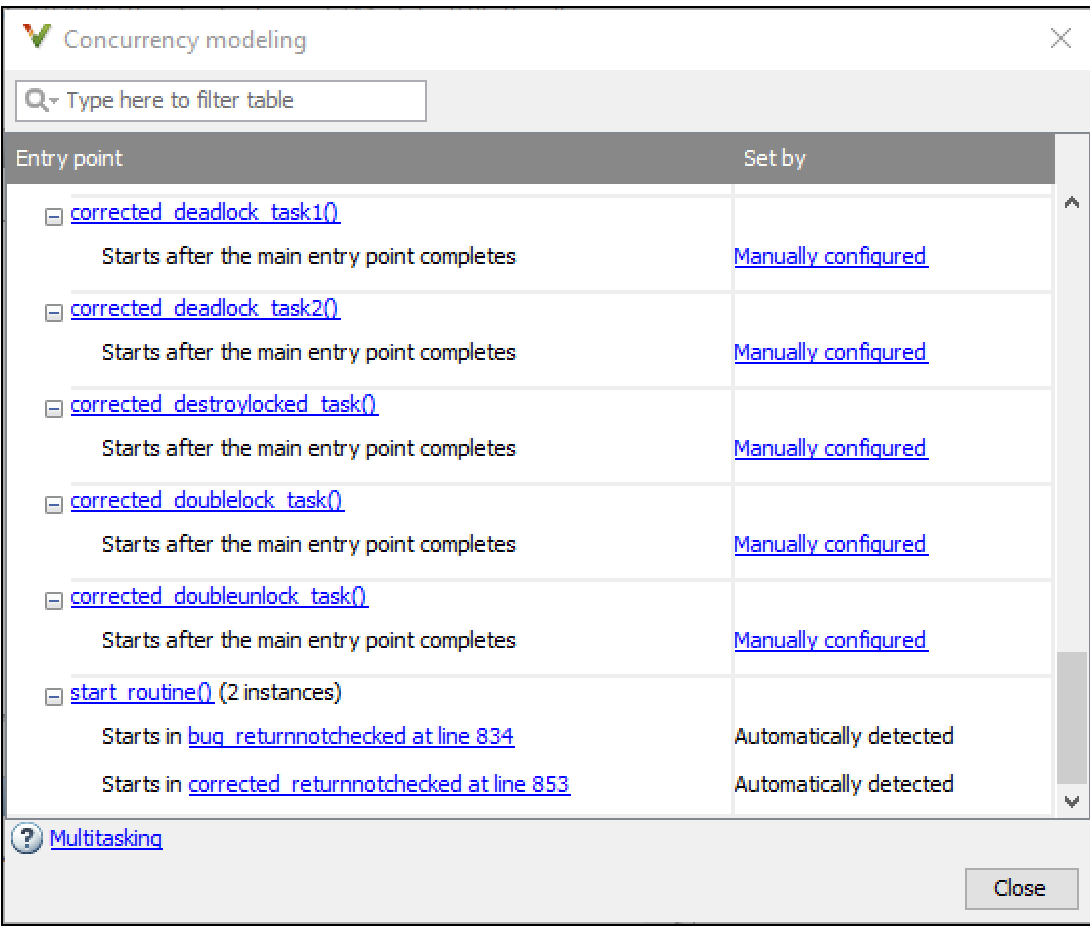
Other Dashboard Features

You can also perform the following actions on this pane:

- Select elements on the graphs to filter results from the **Results List** pane. See "Filter and Group Results in Polyspace Desktop User Interface" on page 24-2.
- View the configuration used to obtain the result. Select the link **Configuration**.
- View information about functions that are not reached during the analysis. Select the link **Unreachable functions**.
- View the analysis assumptions behind the result. Select the link **Analysis assumptions**.
- View the modeling of the multitasking configuration of your code. Select the link **Concurrency modeling on page 21-11**.

Concurrency Modeling in Polyspace Desktop User Interface

The **Concurrency Modeling** view displays all the tasks and interrupts that the analysis extracts from your code and your Polyspace multitasking configuration.



in the table, the functions are listed in the first column by order of decreasing priority. The second column shows how Polyspace detects each task or interrupt: automatically, manually from the Polyspace configuration, or from an external file.

From this view, you can:

- Click a function name to go to its definition in the source code.
- Click an event to go to the corresponding call to the concurrency primitive in the source code, for instance `pthread_create`.
- Click **Manually configured**, for functions that are manually configured, to go to the **Multitasking** node on the **Configuration** pane.

Results List in Polyspace Desktop User Interface

This topic focuses on the Polyspace desktop user interface. To learn about the equivalent pane in the Polyspace Access web interface, see "Results List in Polyspace Access Web Interface" on page 26-17.

The **Results List** pane lists all analysis results along with their attributes.

For each result, the **Results List** pane contains the check attributes, listed in columns:

Attribute	Description
Family	Group to which the result belongs, for instance, red check, gray check, etc.
ID	Unique identification number of the result.
Type	Result information such as run-time check color (red, orange, green), coding rule standard (MISRA C: 2004, MISRA C: 2012), etc.
Group	Category of the result, for instance: • For run-time checks: Groups such as static memory, numerical, control flow, etc. • For coding rule violations: Groups defined by the coding rule standard. For instance, MISRA C: 2012 defines groups related to code constructs such as functions, pointers and arrays, etc.
Check	Result name, for instance: • For run-time checks: Check name • For coding rule violations: Coding rule number
Detail	Additional information about a result. The column shows the first line of the **Result Details** pane. For an example of how to use this column, see the result `MISRA C:2012 Dir 1.1`.
Information	For orange checks, this column indicates whether the check is related to path or input values. For more information, see "Critical Orange Checks in Polyspace Code Prover" on page 33-9. For coding rule violations, this column indicates whether the rule belongs to the `Required` subset. For global variables, this column contains the global variable name.
File	File containing the instruction where the result occurs

Attribute	Description
Class	Class containing the instruction where the result occurs. If the result is not inside a class definition, then this column contains the entry, **Global Scope**.
Function	Function containing the instruction where the result occurs. If the function is a method of a class, it appears in the format `class_name::function_name`.
Folder	Path to the folder that contains the source file with the result
Line	Line number of the instruction where the result occurs.
Col	Column number of the instruction where the result occurs. The column number is the number of characters from the beginning of the line.
%	Percentage of run-time checks that are not orange (total selectivity rate). This column is most useful when you choose the option **File** from the list. The entry in this column against a file or function indicates the percentage of checks in the file or function that are not orange.
Severity	Level of severity you have assigned to the result. The possible levels are: • `Unset` • `High` • `Medium` • `Low`
Status	Review status you have assigned to the result. The possible statuses are: • `Unreviewed` (default status) • `To investigate` • `To fix` • `Justified` • `No action planned` • `Not a defect` • `Other`

Attribute	Description
Justified	Check boxes showing whether you have justified the results. To justify a result, you must assign the status `Justified`, `No action planned` or `Not a defect`. If you choose the option **File** from the [icon] list, this column indicates the percentage of checks that you have justified per file and function.
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”.

Source Code in Polyspace Desktop User Interface

This topic focuses on the Polyspace desktop user interface. To learn about the equivalent pane in the Polyspace Access web interface, see “Source Code in Polyspace Access Web Interface” on page 26-19.

The **Source** pane shows the source code with the results highlighted with specific colors and icons. For more information, see “Code Prover Result and Source Code Colors” on page 32-2.

Source

Dashboard main.c

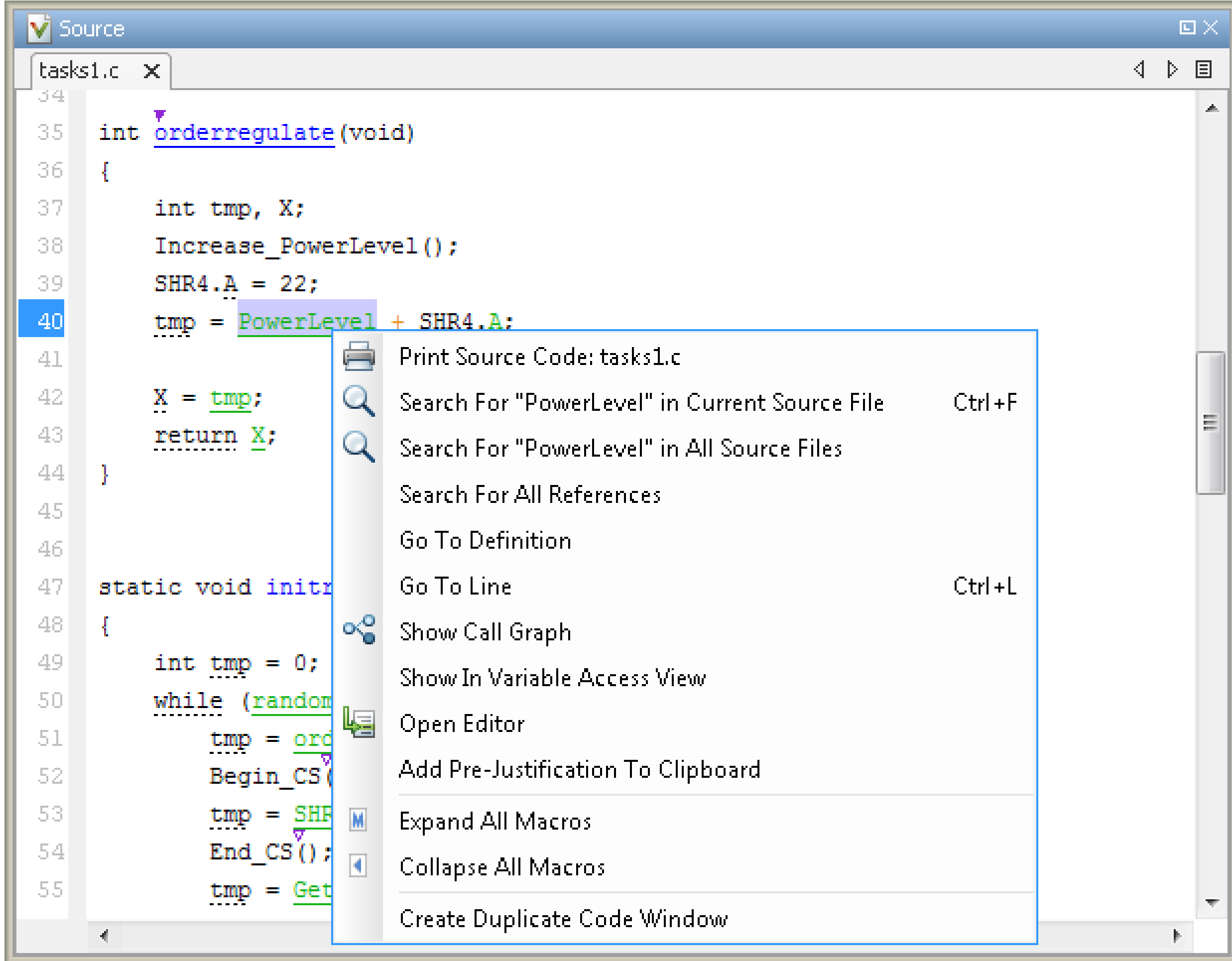
```
13
14
15  static int interpolation(void)
16  {
17      int i, item = 0;
18      int found = false;
19
20
21      for (i = 0; i < MAX_SIZE; i++) {
22          arr++;
23          if ((found == false) && (*arr > 16)) {
24              found = true;
25              item = i;
26          }
27      }
28      *arr = 20;
29      return item;
30  }
31
```

Source | Data Range Configuration

On the **Source** pane, you can perform the following actions.

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, right-click the global variable `PowerLevel`:



Use the following options to examine and navigate through your code:

- **Search "PowerLevel" in Current Source File** — List occurrences of the string within the current source file in the **Search** pane.
- **Search "PowerLevel" in All Source Files** — List occurrences of the string within all source files in 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 `PowerLevel`. The software supports this feature for global and local variables, functions, types, and classes. If the definition is not available to Polyspace, selecting the option takes you to the function 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.

View Variable Ranges

Place your cursor over a check to view range information for variables, operands, function parameters, and return values.

If a tooltip is available for a keyword or identifier on the **Source** pane, Polyspace:

- Uses solid underlining for the keyword or identifier if it is associated with a check.
- Uses dashed underlining for the keyword or identifier if it is not associated with a check.

```
167  static void Square_Root_conv(double alpha, float* beta_pt)
168  /* Perform arithmetic conversion of alpha to beta */
169  {
170      *beta_pt = (float)((1.5 + cos(alpha)) / 5.0);
171  }
172
173
174  stati
175  {
176      d
177      f
178      f
179
180      Square_Root_conv(alpha, &beta);
181
182      gamma = (float)sqrt(beta - 0.75);   /* always sqrt(negative number) */
183  }
```

```
Dereference of parameter 'beta_pt' (pointer to float 32, size: 32 bits):
    Pointer is not null.
    Points to 4 bytes at offset 0 in buffer of 4 bytes, so is within bounds (if memory is allocated).
    Pointer may point to variable or field of variable:
        'beta', local to function 'Square_Root'.
Assignment to dereference of parameter 'beta_pt' (float 32): [0.1 .. 0.5]
                                                                  Press 'F2' for focus
```

The range displayed is the same as the range that the software calculates during verification (or *includes* the range if rounded during display). For instance, for floating point variables, the tooltips show the variable range using the following rules:

- The range appears as a collection of values, for instance `1.0 or 2.0 or NaN`, or an interval `[1.0 .. 2.0]`.
- The displayed range *includes* the actual variable range. For instance, the range `[1.0 .. 2.0]` on a variable indicates that the variable cannot have the value `0.9999` or `2.0001`.

  However, the displayed range can also include additional values because of approximation.
- Constants are displayed using either fixed point (`1.0, -2.0`, etc.) or scientific format when it improves readability (`1.0E+10, -1.2E-20`, etc.).
- The tooltips clearly indicate which values are shown with rounding. For instance, the value `1.0` does not involve rounding but `1.2345...` shows a variable that is displayed with rounding towards zero.

  When rounded, at least 5 significant digits are displayed.

View Extent of Code Block

On the **Source** pane, to highlight a block of code, click either its opening or closing brace.

Source

Dashboard | tasks1.c

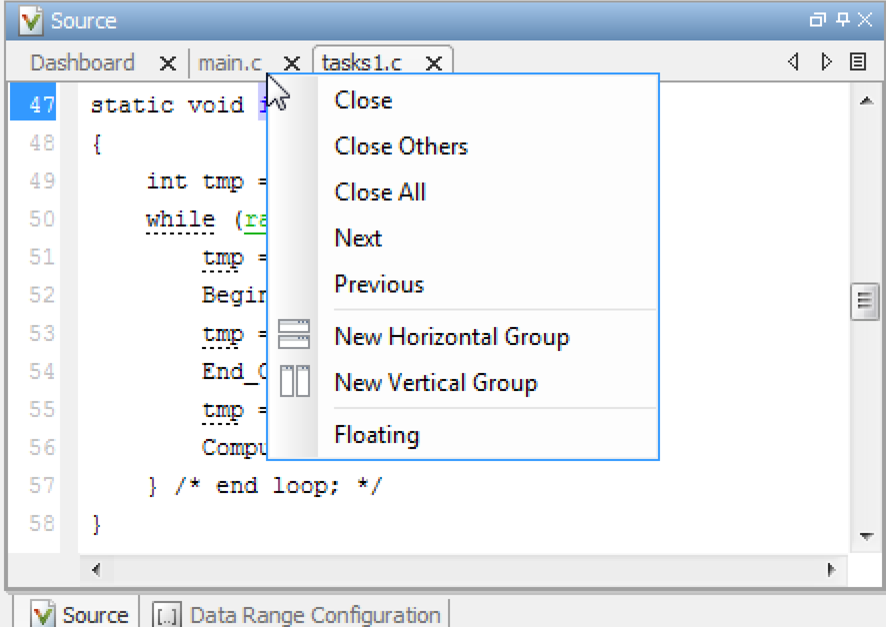
```
47  static void initregulate(void)
48  {
49      int tmp = 0;
50      while (random_int() < 1000) {
51          tmp = orderregulate();
52          Begin_CS();
53          tmp = SHR + SHR2 + SHR6;
54          End_CS();
55          tmp = Get_PowerLevel();
56          Compute_Injection();
57      } /* end loop; */
58  }
```

Source | Data Range Configuration

Manage Multiple Files

You can view multiple source files in the **Source** pane as separate tabs.

On the **Source** pane toolbar, right-click a view.



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

- **Close** – Close the currently selected source file. You can also use the χ button to close the 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** pane horizontally to display the selected source file below another file.
- **New Vertical Group** – Split the **Source** pane 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.

Expand and Collapse Macros

You can view the contents of source code macros in the source code view. A code information bar displays `M` 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.

To display the normal source code again, click the line away from the shaded region, for example, on the arrow icon.

To display or hide the content of *all* macros:

1 Right-click any point within the source code view.
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.

See Function Callers and Callees

You can click on a function name to see callers and callees of the function on the **Call Hierarchy** pane.

- When a function is defined, the source code shows the function name in blue. Click the function name to update the **Call Hierarchy** pane.

```
int func(int val) {
    if(val==0)
        return 0;
    return 1;
}
```

Call Hierarchy

Calls	Line
file.func	13
file.main	8

- When a function is called, the function call either shows a run-time check color or not. If the function does not have a run-time check color (see `func2` below), click the function name to update the **Call Hierarchy** pane.

```
int main() {
    int val = INIT_VAL;
    int checkSuccess = func (val);
    if (checkSuccess == 0)
        func2();
}
```

Call Hierarchy

Calls	Line
file.func2	19
file.main	10

If the function has a run-time check color (see `func` above), right-click the function and select **Go To Definition**. The **Call Hierarchy** pane updates to show the callers and callees.

Result Details in Polyspace Desktop User Interface

This topic focuses on the Polyspace desktop user interface. To learn about the equivalent pane in the Polyspace Access web interface, see “Result Details in Polyspace Access Web Interface” on page 26-24.

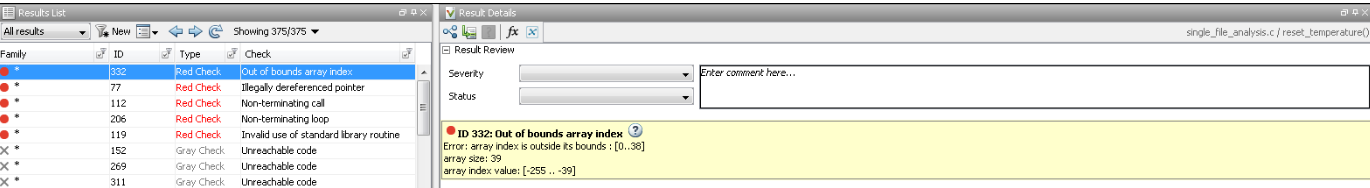
On the **Results List** pane, if you select a check, you see additional information on the **Result Details** pane.

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.

For results that you open from Polyspace Access, you can also:

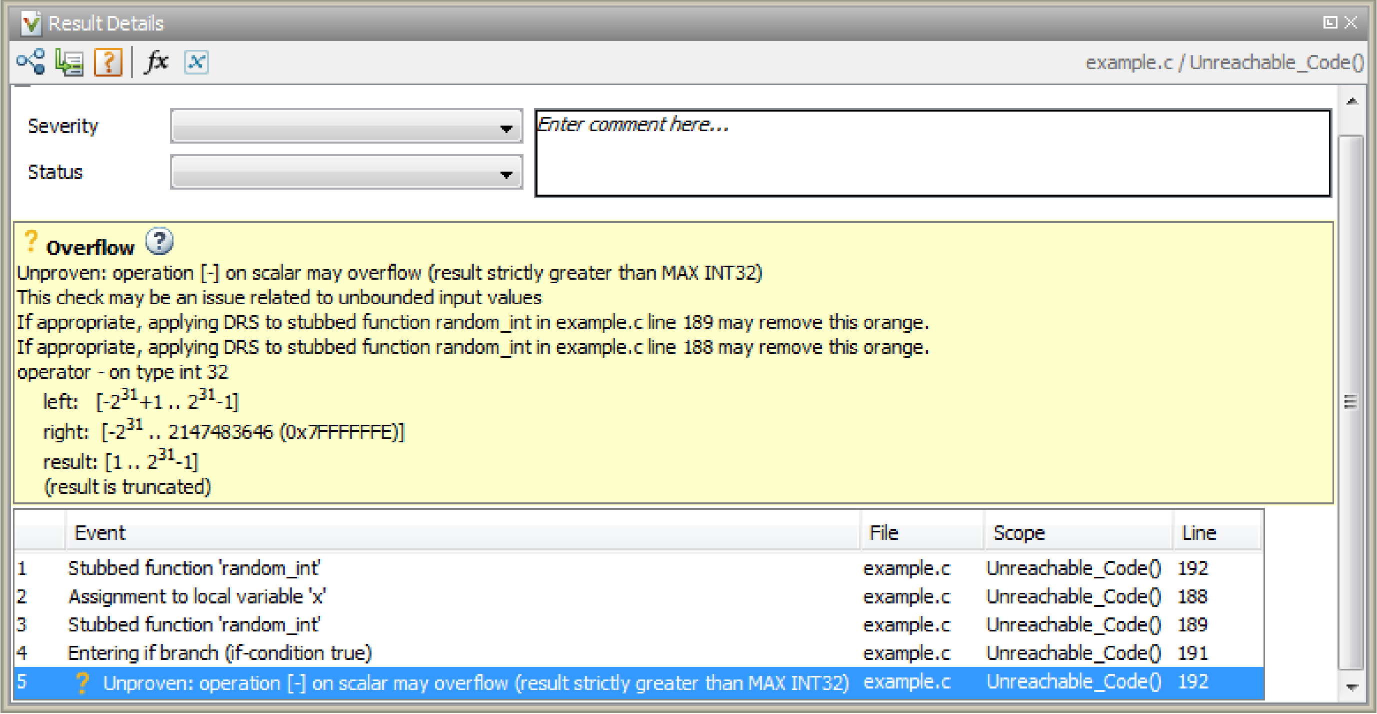
- Assign a reviewer to the result. A reviewer can filter the **Results List** to only show results that are assigned to him or her.
- Create a ticket in a bug tracking tool (BTT) such as JIRA. Once you create the ticket the **Results Details** for this defect shows the ticket ID. Click the ID to open the ticket in the BTT interface.

See “Open or Export Results from Polyspace Access” on page 29-2.



View Traceback

Sometimes, on the **Result Details** pane, you can see the sequence of instructions leading to the check (traceback). You can select each instruction and navigate to it in your source code.

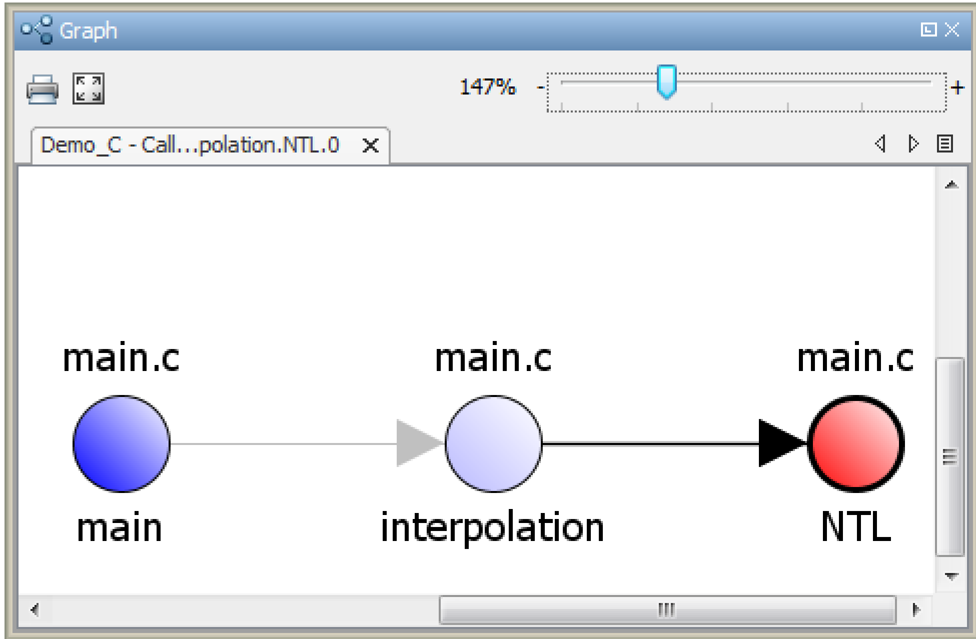


The following columns appear in the traceback:

Column	Description
Event	Code instructions related to the defect. For instance, if an **Out of Bounds Array Index** error occurs in a loop, the **Result Details** pane can show updates to the array index that occur inside the loop. The update statements might physically occur in your code before or after the array access. However, because the statements occur in a loop, they are related to the array access.
Scope	Function containing the instructions. If the instructions are not in a function, the column lists the file containing the instructions.
Line	Line number of the instruction.

View Error Call Graph

Click the **Show error call graph** icon, in the **Result Details** pane toolbar to display the call sequence that leads to the code associated with a result.



For global variables, this graph shows the call sequence leading to read and write operations on the global variable.

View Call Hierarchy and Variable Access

From the **Result Details** pane, you can open the **Call Hierarchy** and **Variable Access** panes.

- Select the *fx* button to open the **Call Hierarchy** pane.

  On this pane, you can see the function in which the current check occurs, along with its callers and callees. For more information, see "Call Hierarchy in Polyspace Desktop User Interface" on page 21-24.

- Select the button to open the **Variable Access** pane.

  On this pane, you can see the global variables in your code. For more information, see "Variable Access in Polyspace Desktop User Interface" on page 21-27.

Call Hierarchy in Polyspace Desktop User Interface

This topic focuses on the Polyspace desktop user interface. To learn about the equivalent pane in the Polyspace Access web interface, see "Call Hierarchy in Polyspace Access Web Interface" on page 26-26.

The **Call Hierarchy** pane displays the call tree of functions in the source code.

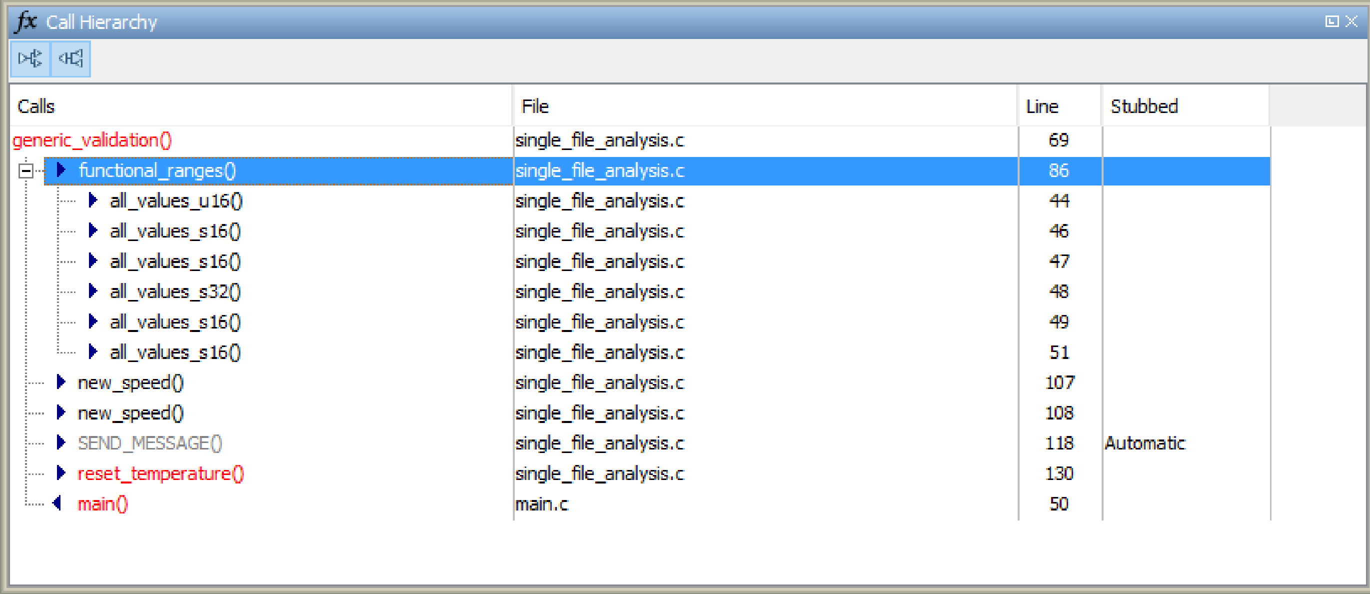
For each function `foo`, the **Call Hierarchy** pane lists the functions and tasks that call `foo` (callers) and those called by `foo` (callees). The callers are indicated by (functions) or (tasks). The callees are indicated by (functions) or (tasks). The **Call Hierarchy** pane lists direct function calls and indirect calls through function pointers. The indirect calls are shown with the icon. Calls that are unreachable are shown with the function name in grey.

To open this pane, in the Polyspace desktop user interface, select the button on the **Result Details** pane.

To update the pane:

- You can click a run-time check, either on the **Results List** or **Source** pane. You see the function containing the check with its callers and callees.
- You can click a function name in your source code. You see the callers and callees of the function. If the function name also shows a run-time check color, instead of clicking the function name, right-click the name and select **Go To Definition**.

In this example, the **Call Hierarchy** pane displays the function `generic_validation`, and its callers and callees.



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.

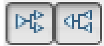
- For a callee name, the number refers to the line where the callee is called. The callee `functional_ranges` is called by `generic_validation` on line 86.
- For a caller name, the number refers to the line where the caller calls the function. The caller `main` calls `generic_validation` on line 50.

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 or Hide Callers and Callees

Customize the view to display callers only or callees only. Show or hide callers and callees by clicking this button:



Navigate Call Hierarchy

You can navigate the call hierarchy in your source code. For a function, double-click a caller or callee name to navigate to the caller or callee definition in the source code.

Determine if Function is Stubbed

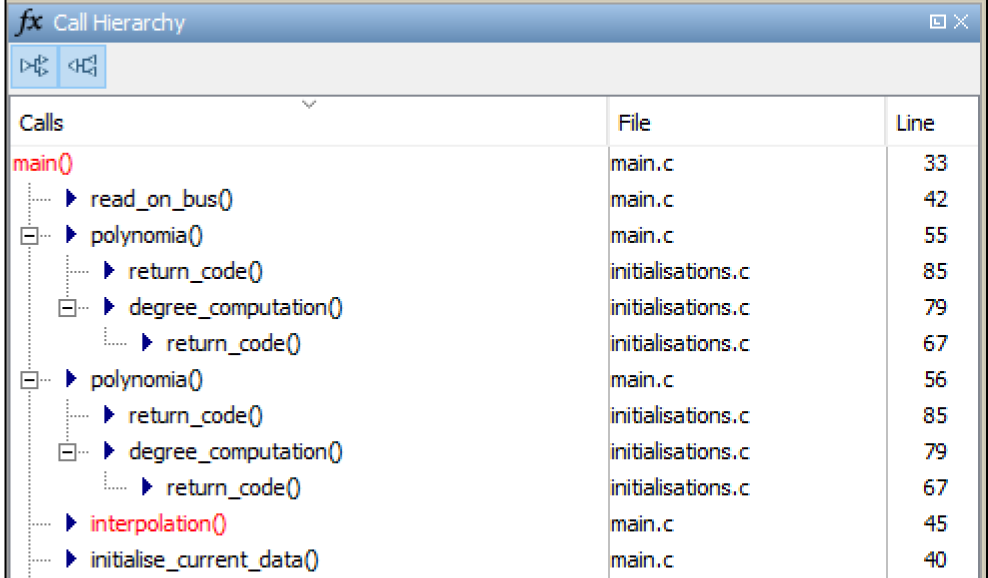
You can determine from the **Stubbed** column if a function is stubbed. The entries in the column show why a function was stubbed.

- **Automatic**: Polyspace cannot find the function definition. For instance, you did not provide the file containing the definition.
- **User specified**: You override the function definition by using the option `Functions to stub (-functions-to-stub)`.
- **Lookup table**: You verify generated code with functions that return values from specific kinds of lookup tables. You use the option `Generate stubs for Embedded Coder lookup tables (-stub-embedded-coder-lookup-table-functions)`.
- **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`.

For more information, see “Code Prover Assumptions About Stubbed Functions”.

See Entire Call Hierarchy of Program

To see the entire call hierarchy of your program, on the **Source** pane, click the `main` function. Right-click a node in the call hierarchy and select **Expand All Nodes**.



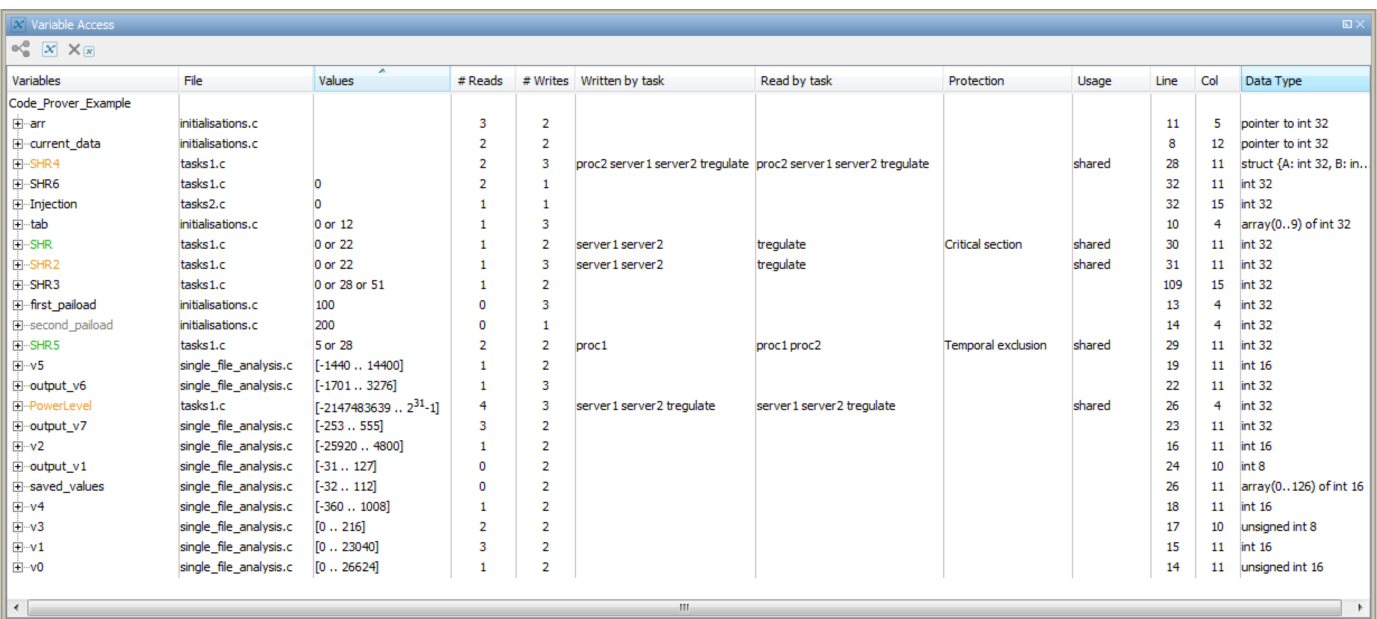
Instead of seeing the entire call hierarchy at once, you can expand nodes as needed to focus on a specific slice of the call hierarchy.

Variable Access in Polyspace Desktop User Interface

This topic focuses on the Polyspace desktop user interface. To learn about the equivalent pane in the Polyspace Access web interface, see “Global Variables in Polyspace Access Web Interface” on page 26-31.

The **Variable Access** pane displays global variables (and local static variables). For each global variable, the pane lists all functions and tasks performing read/write access on the variables, along with their attributes, such as values, read/write accesses and shared usage.

To open this pane, in the Polyspace desktop user interface, select the button on the **Result Details** pane.



Variable Access

Variables	File	Values	# Reads	# Writes	Written by task	Read by task	Protection	Usage	Line	Col	Data Type
Code_Prover_Example											
arr	initialisations.c		3	2					11	5	pointer to int 32
current_data	initialisations.c		2	2					8	12	pointer to int 32
SHR4	tasks1.c		2	3	proc2 server1 server2 tregulate	proc2 server1 server2 tregulate		shared	28	11	struct {A: int 32, B: in..
SHR6	tasks1.c	0	2	1					32	11	int 32
Injection	tasks2.c	0	1	1					32	15	int 32
tab	initialisations.c	0 or 12	1	3					10	4	array(0..9) of int 32
SHR	tasks1.c	0 or 22	1	2	server1 server2	tregulate	Critical section	shared	30	11	int 32
SHR2	tasks1.c	0 or 22	1	3	server1 server2	tregulate		shared	31	11	int 32
SHR3	tasks1.c	0 or 28 or 51	1	2					109	15	int 32
first_paiload	initialisations.c	100	0	3					13	4	int 32
second_paiload	initialisations.c	200	0	1					14	4	int 32
SHR5	tasks1.c	5 or 28	2	2	proc1	proc1 proc2	Temporal exclusion	shared	29	11	int 32
v5	single_file_analysis.c	[-1440 .. 14400]	1	2					19	11	int 16
output_v6	single_file_analysis.c	[-1701 .. 3276]	1	3					22	11	int 32
PowerLevel	tasks1.c	[-2147483639 .. 2^{31}-1]	4	3	server1 server2 tregulate	server1 server2 tregulate		shared	26	4	int 32
output_v7	single_file_analysis.c	[-253 .. 555]	3	2					23	11	int 32
v2	single_file_analysis.c	[-25920 .. 4800]	1	2					16	11	int 16
output_v1	single_file_analysis.c	[-31 .. 127]	0	2					24	10	int 8
saved_values	single_file_analysis.c	[-32 .. 112]	0	2					26	11	array(0..126) of int 16
v4	single_file_analysis.c	[-360 .. 1008]	1	2					18	11	int 16
v3	single_file_analysis.c	[0 .. 216]	2	2					17	10	unsigned int 8
v1	single_file_analysis.c	[0 .. 23040]	3	2					15	11	int 16
v0	single_file_analysis.c	[0 .. 26624]	1	2					14	11	unsigned int 16

For each variable and each read/write access, the **Variable Access** pane contains the relevant attributes. For the variables, the various attributes are listed in this table.

Attribute	Description
Variables	Name of Variable
File	Source file containing variable declaration
Values	Value (or range of values) of variable This column is empty for pointer variables.
# Reads	Number of times the variable is read
# Writes	Number of times the variable is written
Written by task	Name of tasks writing on variable
Read by task	Name of tasks reading variable

Attribute	Description
Protection	Whether shared variable is protected from concurrent access (Filled only when **Usage** column has entry, **Shared**) The possible entries in this column are: • **Critical Section**: If variable is accessed in critical section of code • **Temporal Exclusion**: If variable is accessed in mutually exclusive tasks For more details on these entries, see “Multitasking”.
Usage	`Shared`, if variable is shared between tasks; otherwise, blank
Line	Line number of variable declaration
Col	Column number (number of characters from beginning of line) of variable declaration
Data Type	Data type of variable (C/C++ data types or structures/ classes)

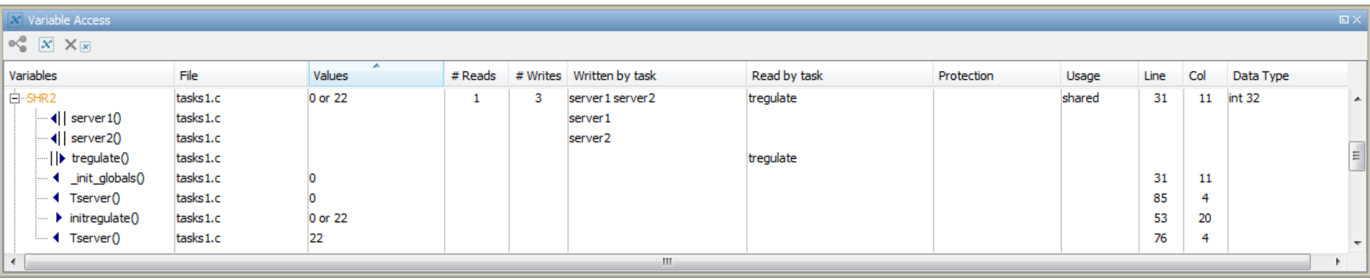
Double-click a variable name to view read/write access operations on the variable. The arrowhead symbols ▶ and ◀ in the **Variable Access** pane indicate functions performing read and write access respectively on the global variable. Likewise, tasks performing read and write access are indicated by the symbols ||▶ and ◀|| respectively. For further information on tasks, see `Tasks (-entry-points)`.

For access operations on the variables, the various attributes described in the pane are listed in this table.

Attribute	Description
Variables	Names of function (or task) performing read/write access on the variable
Values	Value or range of values of variable in the function or task performing read/write access This column is empty for pointer variables.
Written by task	*Only for tasks*: Name of task performing write access on variable
Read by task	*Only for tasks*: Name of task performing read access on variable
Line	Line number where function or task accesses variable
Col	Column number where function or task accesses variable

Attribute	Description
File	Source file containing access operation on variable If this column contains the name `__polyspace__stdstubs.c`, it indicates that the variable is accessed inside a Standard Library function.

For example, consider the global variable, SHR2:



The function, `Tserver`, in the file, `tasks1.c`, performs two write operations on SHR2. This is indicated in the **Variable Access** pane by the two instances of `Tserver()` under the variable, SHR2, marked by . Likewise, the two write accesses by tasks, `server1` and `server2`, are also listed under SHR2 and marked by .

The color scheme for variables in the **Variable Access** pane is:

- Black: global variable.
- Orange: global variable, shared between tasks with no protection against concurrent access.
- Green: global variable, shared between tasks and protected against concurrent access.
- Gray: global variable, declared but not used in reachable code.

If a task performs certain operations on a global variable, but the operations are in unreachable code, the tasks are colored gray.

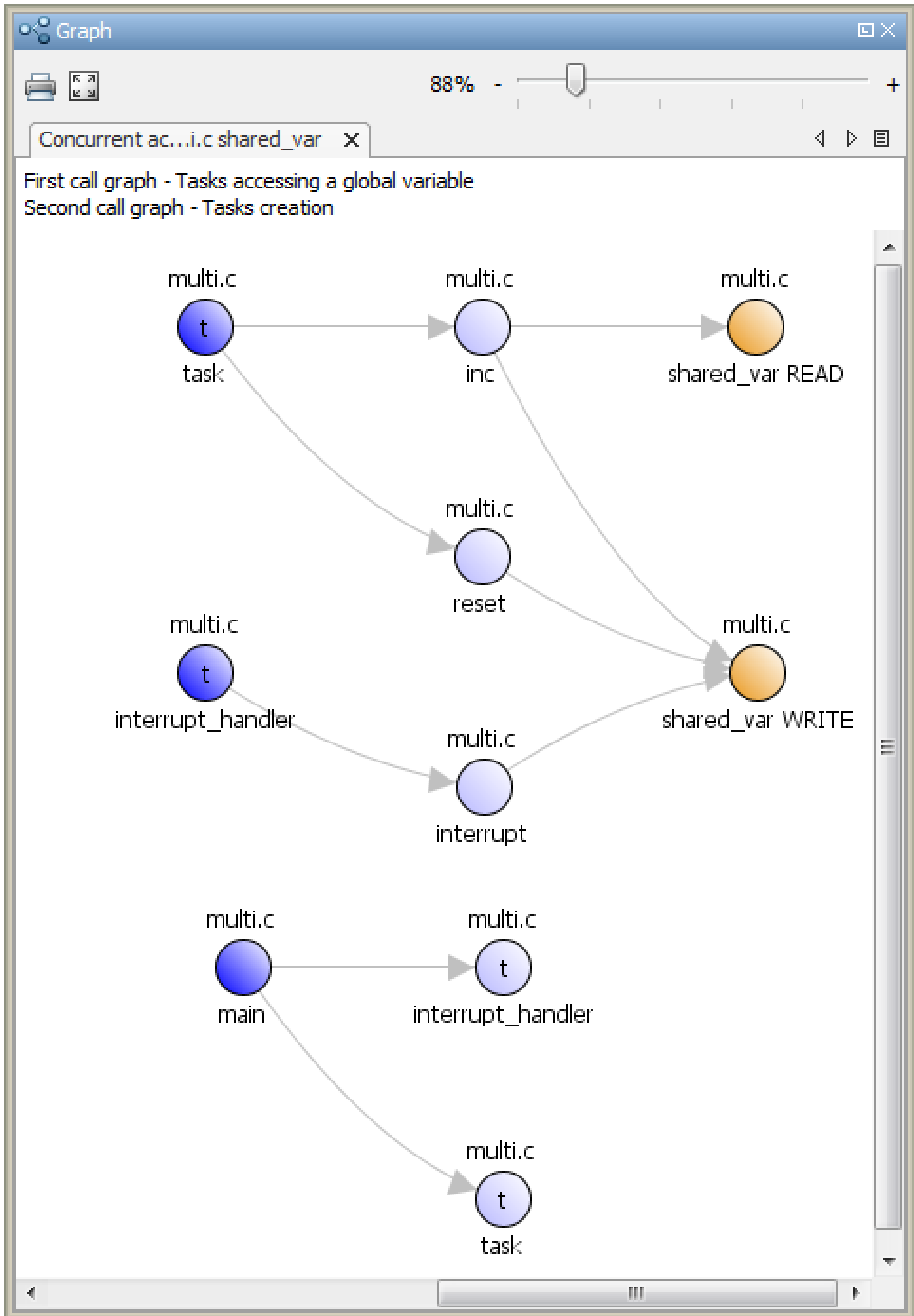
The information about global variables and read/write access operations obtained from the **Variable Access** pane is called the data dictionary.

You can also perform the following actions from the **Variable Access** pane.

View Access Graph

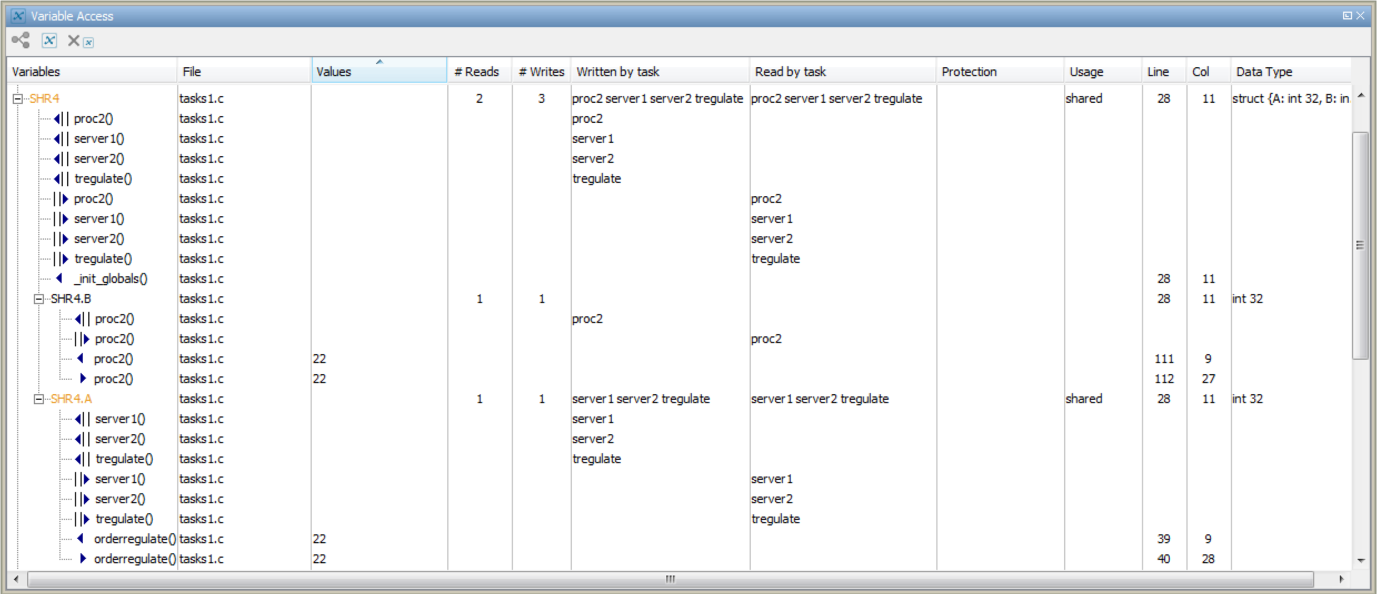
View the access operations on a global variable in graphical format using the **Variable Access** pane. Select the global variable and click .

Here is an example of an access graph:



View Structured Variables

For structured variables, view the individual fields from the **Variable Access** pane. For example, for the structure, SHR4, the pane displays the fields, SHR4.A and SHR4.B, and the functions performing read/write access on them.

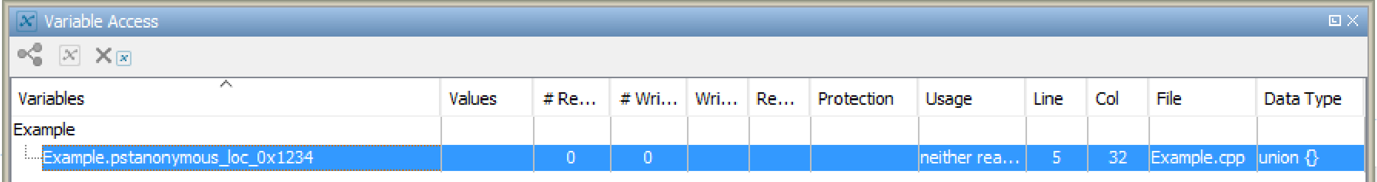


View Operations on Anonymous Variables

You can view operations on anonymous variables. For example, consider this line of code that declares an unnamed union with the variable at an absolute address:

```
union {char, c; int i; } @0x1234;
```

When you analyze the preceding code and specify the `iar` compiler, the unnamed variable at `0x1234` appears in the **Variable Access** pane with a name that starts with **pstanonymous**.

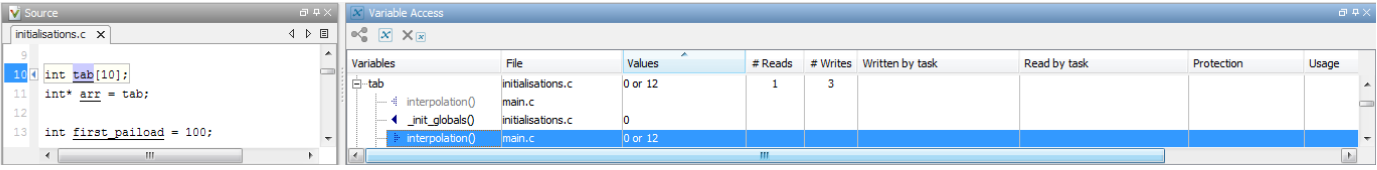


View Access Through Global Pointers

View access operations on global variables performed indirectly through global pointers.

If a read/write access on a variable is performed through global pointers, then the access is marked by (read) or (write). Access through pointers is shown like any other direct access.

For instance, in the file, `initialisations.c`, the variable, `arr`, is declared as a pointer to the array, `tab`.



In the file `main.c`, `tab` is read in the function, `interpolation()`, through the global pointer variable, `arr`. This operation is shown in the **Variable Access** pane by the icon.

During dynamic memory allocation, memory is allocated directly to a pointer. Because the **Values** column is populated only for non-pointer variables, you cannot use this column to find the values stored in dynamically allocated memory. Use the **Variable Access** pane to navigate to dereferences of the pointer on the **Source** pane. Use the tooltips on this pane to find the values following each pointer dereference.

Show or Hide Callers and Callees

Customize the **Variable Access** pane to show only the shared variables. On the **Variable Access** pane toolbar, click the Non-Shared Variables button to show or hide non-shared variables.

Show or Hide Accesses in Unreachable Code

Hide read/write access occurring in unreachable code by clicking the filter button .

Other Features

You cannot see an addressing operation on a global variable or object (in C++) as a read/write operation in the **Variable Access** pane. For example, consider the following C++ code:

```
class C0
{
public:
  C0() {}
  int get_flag()
  {
    volatile int rd;
    return rd;
  }
  ~C0() {}
private:
  int a;                /* Never read/written */
};

C0 c0;                  /* c0 is unreachable */

int main()
{
  if (c0.get_flag())    /* Uses address of the method */
    {
      int *ptr = take_addr_of_x();
      return 1;
    }
  else
    return 0;
}
```

You do not see the method call `c0.get_flag()` in the **Variable Access** pane because the call is an addressing operation on the method belonging to the object `c0`.

Understanding Changes in Polyspace Results After Product Upgrade

This topic describes how to interpret changes in results after upgrading Polyspace Code Prover. For product upgrade instructions, see "Update Polyspace Products" or "Update or Uninstall Polyspace Access".

If you upgrade to a newer release of Polyspace, you can see some changes in results for the same analysis. Each release introduces many improvements in analysis precision. These improvements can lead to the same analysis (same source files and options) showing a difference in results before and after the upgrade.

This topic describes the kinds of differences you might see, why they might be expected and how you can understand those differences. For information on how to compare two sets of results, see "Migrate Polyspace Projects After Product Upgrade".

Changes in Polyspace Code Prover Results

For the same source code and analysis configuration, you might see a change in results because of improvements to the Polyspace Code Prover analysis engine. In Code Prover, a change in result means a change in color for the same run-time check. When comparing results, you can focus only on new red, grey and orange checks. As explained later, new green checks are typically the result of an increase in precision.

- For major differences in results of a specific type, see if you can trace the difference to a documented change in behavior or assumptions.

  Check the release notes of all releases between your prior release and the release you upgraded to. Look in the **Verification results** section in the *Release Notes for Polyspace Code Prover* for changes in behavior of specific checks or changes in Code Prover assumptions. Major changes in behavior or assumptions are typically documented in the release notes.
- For differences in results that cannot be traced to a documented change, see if you can attribute the change in color to an increase in verification precision.

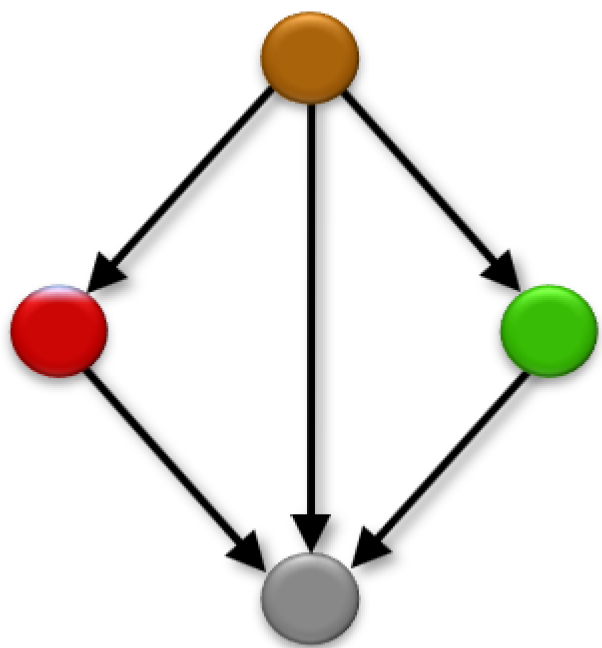
  In addition to documented major changes, each new release also involves many minor improvements in the verification algorithms. These improvements typically lead to an increase in verification precision (or at least maintain the same precision as before).

  Run-time checks in Polyspace Code Prover can return results in one of three colors:

  - Red (proof of definite error)
  - Green (proof of definite absence of error)
  - Red (unproven, probable error)

  In addition, a grey color is used for code that is unreachable and therefore not checked for other run-time errors. For more details, see "Code Prover Result and Source Code Colors" on page 32-2.

  An increase in precision indicates changes in color in the direction shown in this figure:



In other words, one of these things might happen:

- **Orange to green or red**: An operation that shows an orange color might now be red or green. Code Prover has a more precise knowledge of variable ranges when analyzing that operation, so the presence or absence of an error can be proven.

  During verification, to check if an operation causes run-time errors, Code Prover uses previously computed possible values of the operands. Some of these values might be accumulated from the code (from initializations and subsequent write operations along various paths), but some of them are results of approximations and cannot happen in practice. An increase in precision leads to fewer approximations, so fewer values that come solely from approximations. As a result, it is possible to obtain proof in a greater number of cases.

  Consider this simple, illustrative example:

  ```
  x = initValue;
  y = x - 1;

  // Call below increases its second arg if the first is positive
  incr_y_if_x_positive(x,&y);

  // Later operations involving x and y
  interval = x - y ;
  if (x > 0)
     interval = x;
  num = range / interval;
  ```

  In this example, to prove that there is no division by zero, Code Prover has to keep track of the fact that `interval = x - y` can be zero only if `x > 0` (and the fact that the case `x > 0` is handled later). Because of approximations, Code Prover might not be able to keep track of

relations between variables such as `x` and `y` across several lines of code leading to an orange division by zero check. An increase in precision means that Code Prover is able to track such kinds of relations on more complex operations, leading to a green division by zero check.

- **Red, green or orange to grey**: An operation that was previously checked and showed one of red, green or orange colors is now proven unreachable and appears in grey.

  As before, lower precision means considering more values from approximations. So, a conditional branch such as:

  ```
  if (x >= 0)
  ```

  that is unreachable can be considered reachable because of a negative value of `x` coming from approximations. With the increase in precision, this value might no longer be considered making it possible to prove that the branch is unreachable. If the branch is proven unreachable, red, green and orange checks inside the branch disappear.

For an individual check, it is easier to understand this change in direction of colors. However, these changes of colors in individual statements do not translate directly to changes in overall number of results of a certain color and type. For instance, some orange division by zero checks might turn green, but some green checks might also turn grey, leading to an overall increase or decrease in green division by zero checks. Therefore, when comparing results, instead of focusing on changes in overall numbers of checks, compare a few individual checks that changed color. In most cases, you will see that the color change is an expected result of an increase in precision.

In rare cases, you can see a decrease of precision (changes of colors in the opposite direction compared to what is described in this section). Typically, changes in Polyspace algorithms are vetted against a representative database of code samples to make sure that they do not cause significant decrease in precision or increase in analysis time.

Changes in Polyspace Bug Finder Results

For the same source code and analysis configuration, you might see a change in results because of improvements to the Polyspace Bug Finder analysis engine. In Bug Finder, a change in results falls in one of these categories: a new result appears, an existing result no longer shows up, or the same result appears on a different location in the source code.

Suppose that you see a new result in the current release. The new result might appear because of updates to a specific checker or from general updates to the analysis algorithm that affects the result locations for several checkers. To find more details, check the release notes of all releases between your prior release and the release you upgraded to.

- Look in the **Analysis results** section of the *Release Notes for Polyspace Bug Finder* for updates to a specific coding rule or defect checker:

  - Updates to existing defect checkers appear in a specific entry **Updated Bug Finder defect checkers**.
  - Updates to existing external coding standard checkers appear in a specific entry **Changes to external coding standards checking**.
- Look in the **Reviewing results** section of the *Release Notes for Polyspace Bug Finder* for changes in location of checker results. For instance, if a result previously appeared in separate instances of a macro, it might now be rolled up to the macro definition.

See Also

Related Examples

- “Migrate Polyspace Projects After Product Upgrade”

Reviewing Checks

- "Review and Fix Absolute Address Usage Checks" on page 22-2
- "Review and Fix Correctness Condition Checks" on page 22-3
- "Review and Fix Division by Zero Checks" on page 22-7
- "Review and Fix Function Not Called Checks" on page 22-11
- "Review and Fix Function Not Reachable Checks" on page 22-13
- "Review and Fix Function Not Returning Value Checks" on page 22-15
- "Review and Fix Illegally Dereferenced Pointer Checks" on page 22-17
- "Review and Fix Incorrect Object Oriented Programming Checks" on page 22-22
- "Review and Fix Invalid C++ Specific Operations Checks" on page 22-24
- "Review and Fix Invalid Shift Operations Checks" on page 22-26
- "Review and Fix Invalid Use of Standard Library Routine Checks" on page 22-30
- "Invalid Use of Standard Library Floating Point Routines" on page 22-32
- "Review and Fix Non-initialized Local Variable Checks" on page 22-35
- "Review and Fix Non-initialized Pointer Checks" on page 22-38
- "Review and Fix Non-initialized Variable Checks" on page 22-40
- "Review and Fix Non-Terminating Call Checks" on page 22-42
- "Identify Function Call with Run-Time Error" on page 22-44
- "Review and Fix Non-Terminating Loop Checks" on page 22-46
- "Identify Loop Operation with Run-Time Error" on page 22-49
- "Review and Fix Null This-pointer Calling Method Checks" on page 22-51
- "Review and Fix Out of Bounds Array Index Checks" on page 22-53
- "Review and Fix Overflow Checks" on page 22-57
- "Detect Overflows in Buffer Size Computation" on page 22-61
- "Review and Fix Return Value Not Initialized Checks" on page 22-63
- "Review and Fix Uncaught Exception Checks" on page 22-66
- "Review and Fix Unreachable Code Checks" on page 22-68
- "Review and Fix User Assertion Checks" on page 22-73
- "Find Relations Between Variables in Code" on page 22-77
- "Review Polyspace Results on AUTOSAR Code" on page 22-80

Review and Fix Absolute Address Usage Checks

This topic describes how to systematically review the results of an **Absolute address usage** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Absolute address usage** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Absolute address usage`.

Tip This check is green by default. To reduce the number of orange checks, if you trust that all absolute addresses in your code are valid, you can retain this default behavior.

For best use of this check, leave this check green by default during initial stages of development. During integration stage, use the option `-no-assumption-on-absolute-addresses` and detect all uses of absolute memory addresses. Browse through them and make sure that the addresses are valid.

---

1. Select the check on the **Results List** pane.

   The **Source** pane displays the code operation containing the absolute address.
2. If you determine that the address is valid, add a comment and justification in your result or code. See "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" on page 23-2 or "Address Results in Polyspace Access Through Bug Fixes or Justifications" on page 27-2.

Review and Fix Correctness Condition Checks

This topic describes how to systematically review the results of a **Correctness condition** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Correctness condition** check. There are multiple ways to fix a red or orange check. For a description of the check and code examples, see `Correctness condition`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

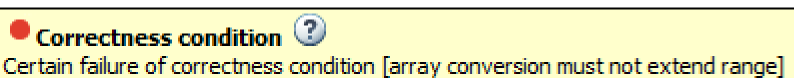
For the general workflow that applies to all checks, see "Interpret Code Prover Results in Polyspace Desktop User Interface" on page 21-2 or "Interpret Code Prover Results in Polyspace Access Web Interface" on page 26-2.

Step 1: Interpret Check Information

On the **Results List** pane, select the check. View the cause of check on the **Result Details** pane. The following list shows some of the possible causes:

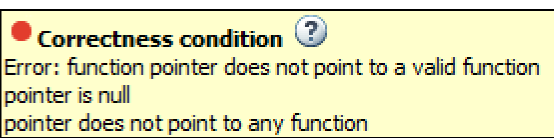
- An array is converted to another array of larger size.

  In the following example, a red check occurs because an array is converted to another array of larger size.

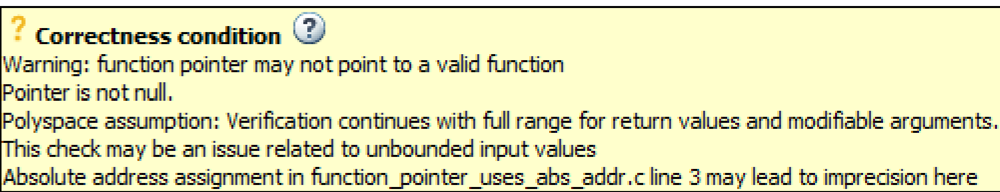
- When dereferenced, a function pointer has value `NULL`.

  In the following example, a red check occurs because, when dereferenced, a function pointer has value `NULL`.

- When dereferenced, a function pointer does not point to a function.

  In the following example, an orange check occurs because Polyspace cannot determine if a function pointer points to a function when dereferenced. This situation can occur if, for instance, you assign an absolute address to the function pointer.

- A function pointer points to a function, but the argument types of the pointer and the function do not match. For example:

```
typedef int (*typeFuncPtr) (complex*);
int func(int* x);
.
.
typeFuncPtr funcPtr = &func;
```

  In the following example, a red check occurs because:

  - The function pointer points to a function `func`.
  - `func` expects an argument of type `int`, but the corresponding argument of the function pointer is a structure.

Correctness condition
Warning: function pointer may not point to a valid function
Pointer is not null.
Pointer points to badly typed function: func.
- Error when calling function func: wrong type of argument (argument 1 of call has type pointer to structure but function expects type pointer to int 32).
Polyspace assumption: Verification continues with full range for return values and modifiable arguments.

- A function pointer points to a function, but the argument numbers of the pointer and the function do not match. For example:

```
typedef int (*typeFuncPtr) (int, int);
int func(int);
.
.
typeFuncPtr funcPtr = &func;.
```

  In the following example, a red check occurs because:

  - The function pointer points to a function `func`.
  - `func` expects one argument but the function pointer has two arguments.

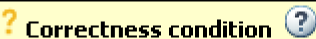
Correctness condition
Warning: function pointer may not point to a valid function
Pointer is not null.
Pointer points to badly typed function: func.
- Error when calling function func: wrong number of arguments (call has 2 arguments but function expects 1 argument).
Polyspace assumption: Verification continues with full range for return values and modifiable arguments.

- A function pointer points to a function, but the return types of the pointer and the function do not match. For example:

```
typedef double (*typeFuncPtr) (int);
int func(int);
.
.
typeFuncPtr funcPtr = &func;
```

  In the following example, a red check occurs because:

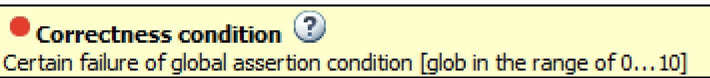
  - The function pointer points to a function `func`.
  - `func` returns an `int` value, but the return type of the function pointer is `double`.



- The value of a variable falls outside the range that you specify through the **Global Assert** mode. See “Constrain Global Variable Range for Polyspace Analysis” on page 14-12.

  In the following example, a red check occurs because:

  - You specify a range 0...10 for the variable `glob`.
  - The value of the variable falls outside this range.



Step 2: Determine Root Cause of Check

Based on the check information on the **Result Details** pane, perform further steps to determine the root cause. You can perform the following steps in the Polyspace user interface only.

Check Information	How to Determine Root Cause
An array is converted to another array of larger size.	**1** To determine the array sizes, see the definition of each array variable. Right-click the variable and select **Go To Definition**. **2** If you dynamically allocate memory to an array, it is possible that their sizes are not available during definition. Browse through all instances of the array variable to find where you allocate memory to the array. **a** Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. **b** On the **Search** pane, select the previous instances.

<table>
<tr><th>Check Information</th><th>How to Determine Root Cause</th></tr>
<tr><td>Issues when dereferencing a function pointer:

• The function pointer has value NULL when dereferenced.
• The function pointer does not point to a function when dereferenced.
• The function pointer points to a function, but the argument types of the pointer and the function do not match.
• The function pointer points to a function, but the argument numbers of the pointer and the function do not match.
• The function pointer points to a function, but the return types of the pointer and the function do not match.</td><td>1 Find the location where you assign the function pointer to a function.

a Right-click the function pointer. Select Search For All References.

All instances of the function pointer appear on the Search pane with the current instance highlighted.
b On the Search pane, select the previous instances.
2 Determine the argument and return types of the function pointer type and the function. Identify if there is a mismatch between the two. For instance, in the following example, determine the argument and return types of <code>typeFuncPtr</code> and <code>func</code>.

<code>typeFuncPtr funcPtr = func;</code>

a Right-click the function pointer type and select Go To Definition.
b Right-click the function and select Go To Definition. If the definition does not exist, this option shows the function stub definition instead. In this case, find the function declaration.
3 Sometimes, you assign a function pointer to a function with matching signature, but the assignment is unreachable. Check if this is the case.</td></tr>
<tr><td>The value of a variable falls outside the range that you specify through the Global Assert mode.</td><td>Browse through all previous instances of the global variable. Identify a suitable point to constrain the variable.

1 Right-click the variable. Select Show In Variable Access View.
2 On the Variable Access pane, select each instance of the variable.</td></tr>
</table>

Step 3: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2 or “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2.

Review and Fix Division by Zero Checks

This topic describes how to systematically review the results of a **Division by zero** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Division by zero** check. There are multiple ways to fix a red or orange check. For a description of the check and code examples, see `Division by zero`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2 or “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2.

Step 1: Interpret Check Information

Place your cursor on the `/` or `%` operation that causes the **Division by zero** error.

```
func(1.0/val);
```

Probable cause for 'Division by Zero': Stubbed function 'getVal'
operator / on type float 64
left: 1.0
right: $[-2.1475E^{+09} .. 2.1475E^{+09}]$
result: $[-1.0001 .. -4.6566E^{-10}]$ or $[4.6566E^{-10} .. 1.0001]$

Obtain the following information from the tooltip:

- The values of the right operand (denominator).

  In the preceding example, the right operand, `val`, has a range that contains zero.

  Possible fix: To avoid the division by zero, perform the division only if `val` is not zero.

Integer	Floating-point

`if(val != 0)`
`    func(1.0/val);`
`else`
`    /* Error handling */`
	`#define eps 0.0000001`
	`.`
	`.`
	`if(val < -eps || val > eps)`
	`    func(1.0/val);`
	`else`
	`    /* Error handling */`

- The probable root cause for division by zero, if indicated in the tooltip.

  In the preceding example, the software identifies a stubbed function, `getVal`, as probable cause.

Possible fix: To avoid the division by zero, constrain the return value of `getVal`. For instance, specify that `getVal` returns values in a certain range, for example, `1..10`. To specify constraints, use the analysis option `Constraint setup (-data-range-specifications)`.

Step 2: Determine Root Cause of Check

Before a `/` or `%` operation, test if the denominator is zero. Provide appropriate error handling if the denominator is zero.

Only if you do not expect a zero denominator, determine root cause of check. Trace the data flow starting from the denominator variable. Identify a point where you can specify a constraint to prevent the zero value.

In the following example, trace the data flow starting from `arg2`:

```
void foo() {
    double time = readTime();
    double dist = readDist();
    .
    .
  bar(dist,time);
}

void bar(double arg1, double arg2) {
    double vel;
    vel=arg1/arg2;
}
```

You might find that:

1. `bar` is called with full-range of values.

   Possible fix: Call `bar` only if its second argument `time` is greater than zero.
2. `time` obtains a full-range of values from `readTime`.

   Possible fix: Constrain the return value of `readTime`, either in the body of `readTime` or through the Polyspace Constraint Specification interface, if you do not have the definition of `readTime`. For more information, see “Code Prover Assumptions About Stubbed Functions”.

To trace the data flow, select the check and note the information on the **Result Details** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Result Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.
- Otherwise:
  1. Find the previous write operation on the operand variable.

     Example: The value of `arg2` is written from the value of `time` in `bar`.
  2. At the previous write operation, identify a new variable to trace back.

     Place your cursor on the variables involved in the write operation to see their values. The values help you decide which variable to trace.

Example: At `bar(dist,time)`, you find that `time` has a full-range of values. Therefore, you trace `time`.

3. Find the previous write operation on the new variable. Continue tracing back in this way until you identify a point to specify your constraint.

   Example: The previous write operation on `time` is `time=readTime()`. You can choose to specify your constraint on the return value of `readTime`.

Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

<table>
<tr><th>Variable</th><th>How to Find Previous Instances of Variable</th></tr>
<tr><td>Local Variable</td><td>Use one of the following methods:
• Search for the variable.
1. Right-click the variable. Select Search For All References.
All instances of the variable appear on the Search pane with the current instance highlighted.
2. On the Search pane, select the previous instances.
• Browse the source code.
1. Double-click the variable on the Source pane.
All instances of the variable are highlighted.
2. Scroll up to find the previous instances.</td></tr>
<tr><td>Global Variable

Right-click the variable. If the option Show In Variable Access View appears, the variable is a global variable.</td><td>1. Select the option Show In Variable Access View.
On the Variable Access pane, the current instance of the variable is shown.
2. On this pane, select the previous instances of the variable.
Write operations on the variable are indicated with and read operations with .</td></tr>
<tr><td>Function return value

<code>ret=func();</code></td><td>1. Find the function definition.
Right-click <code>func</code> on the Source pane. Select Go To Definition, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration.
2. In the definition of <code>func</code>, identify each <code>return</code> statement. The variable that the function returns is your new variable to trace back.</td></tr>
</table>

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 22-77.

Step 3: Look for Common Causes of Check

Look for common causes of the **Division by zero** check.

- For a variable that you expect to be non-zero, see if you test the variable in your code to exclude the zero value.

  Otherwise, Polyspace cannot determine that the variable has non-zero values. You can also specify constraints outside your code. See “Specify External Constraints for Polyspace Analysis” on page 14-2.
- If you test the variable to exclude its zero value, see if the test occurs in a reduced scope compared to the scope of the division.

  For example, a statement `assert(var !=0)` occurs in an `if` or `while` block, but a division by `var` occurs outside the block. If the code does not enter the `if` or `while` block, the `assert` does not execute. Therefore, outside the `if` or `while` block, Polyspace assumes that `var` can still be zero.

  Possible fix:

  - Investigate why the test occurs in a reduced scope. In the above example, see if you can place the statement `assert(var !=0)` outside the `if` or `for` block.
  - If you expect the `if` or `while` block to always execute, investigate when it does not execute.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2 or “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2.

For instance, you are using a volatile variable in your code. Then:

1. Polyspace assumes that the variable is full-range at every step in the code. The range includes zero.
2. A division by the variable can cause **Division by zero** error.
3. If you know that the variable takes a non-zero value, add a comment and justification describing why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Note Before justifying an orange check, consider carefully whether you can improve your coding design.

Disabling This Check

You can effectively disable this check. If your compiler supports infinities and NaNs from floating-point operations, you can enable a verification mode that incorporates infinities and NaNs. See `Consider non finite floats (-allow-non-finite-floats)`.

Review and Fix Function Not Called Checks

This topic describes how to systematically review the results of a **Function not called** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Function not called** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Function not called`.

If you determine that the check represents defensive code or a function that is part of a library, add a comment and justification in your result or code explaining why you did not change your code. See "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" on page 23-2 or "Address Results in Polyspace Access Through Bug Fixes or Justifications" on page 27-2.

Note This check is not turned on by default. To turn on this check, you must specify the appropriate analysis option. For more information, see `Detect uncalled functions (-uncalled-function-checks)`.

Step 1: Interpret Check Information

On the **Results List** pane, select the check. On the **Source** pane, the body of the function is highlighted in gray.

Source — single_file_analysis.c

```
131
132  static s32 unused_fonction(void)
133  {
134      return saved_values[output_v1] + (s32) generic_validation((output_v6 / 10000) , (output_v7 / 16000));
135  }
```

Step 2: Determine Root Cause of Check

1. Search for the function name and see if you can find a call to the function in your code.

   On the **Search** pane, enter the function name. From the drop-down list beside the search field, select **Source**.

   Possible fix: If you do not find a call to the function, determine why the function definition exists in your code.
2. If you find a call to the function, see if it occurs in the body of another uncalled function.

   Possible fix: Investigate why the latter function is not called.
3. See if you call the function indirectly, for example, through function pointers.

   If the indirection is too deep, Polyspace sometimes cannot determine that a certain function is called.

Possible fix: If Polyspace cannot determine that you are calling a function indirectly, you must verify the function separately. You do not need to write a new `main` function for this other verification. Polyspace can generate a `main` function if you do not provide one in your source. You can change the `main` generation options if needed. For more information on the options, see "Code Prover Verification".

Step 3: Look for Common Causes of Check

Look for the following common causes of the **Function not called** check.

- Determine if you intended to call the function but used another function instead.
- Determine if you intended to replace some code with a function call. You wrote the function definition, but forgot to replace the original code with the function call.

  If this situation occurs, you are likely to have duplicate code.
- See if you intend to call the function from yet unwritten code. If so, retain the function definition.
- For code intended for multitasking, see if you have specified all your entry point functions.

  To see the options used for the result, select the link **View configuration for results** on the **Dashboard** pane.

  For more information, see `Tasks (-entry-points)`.
- For code intended for multitasking, see if your `main` function contains an infinite loop. Polyspace Code Prover requires that your `main` function must complete execution before the other entry points begin.

  For more information, see "Configuring Polyspace Multitasking Analysis Manually" on page 15-17.

Review and Fix Function Not Reachable Checks

This topic describes how to systematically review the results of a **Function not reachable** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Function not reachable** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Function not reachable`.

If you determine that the check represents defensive code, add a comment and justification in your result or code explaining why you did not change your code. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2 or “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2.

Note This check is not turned on by default. To turn on this check, you must specify the appropriate analysis option. For more information, see `Detect uncalled functions (-uncalled-function-checks)`.

Step 1: Interpret Check Information

Select the check on the **Results List** pane. On the **Source** pane, you can see the function definition in gray.

Step 2: Determine Root Cause of Check

Determine where the function is called and review why all the function call sites are unreachable. You can perform the following steps in the Polyspace user interface only.

1. Select the check on the **Results List** pane.
2. On the **Result Details** pane, click the *fx* button.

   On the **Call Hierarchy** pane, you see the callers of the function denoted by ◀.
3. On the **Call Hierarchy** pane, select each caller.

   This action takes you to the function call on the **Source** pane.

4 See if the caller itself is called from unreachable code. If the caller definition is entirely in gray on the **Source** pane, it is called from unreachable code. Follow the same investigation process, starting from step 1, for the caller.

5 Otherwise, investigate why the section of code from which you call the function is unreachable.

   The code can be unreachable because it follows a red check or because it contains the gray **Unreachable code** check.

   - If a red check occurs, fix your code to remove the check.
   - If a gray **Unreachable code** check occurs, review the check and determine if you must fix your code. See “Review and Fix Unreachable Code Checks” on page 22-68.

Note If you do not see a caller name on the **Call Hierarchy** pane, determine if you are calling the function indirectly, for example through a function pointer. Determine if a mismatch occurs between the function pointer declaration and the function call through the pointer.

Polyspace places a red or orange **Correctness condition** check on the indirect call if a mismatch occurs. To detect a mismatch in indirect function calls, look for the **Correctness condition** check on the **Results List** pane. For more information, see `Correctness condition`.

Review and Fix Function Not Returning Value Checks

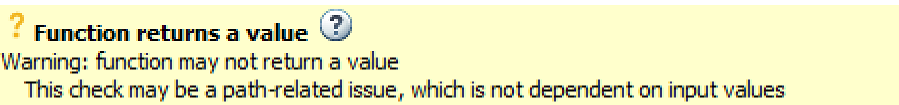
This topic describes how to systematically review the results of a **Function not returning value** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Function not returning value** check. For a description of the check and code examples, see `Function not returning value`.

For the general workflow that applies to all checks, see “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2 or “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2.

Step 1: Interpret Check Information

Select the check on the **Results List** pane. The **Result Details** pane displays further information about the check.



You can see:

- The immediate cause of the check.

  In this example, the software has identified that a function with a non-`void` return type might not have a `return` statement.
- The probable root cause of the check, if indicated.

  In this example, the software has identified that the check is possibly path-related. More than one call to the function exists, and the check is green on at least one call.

Step 2: Determine Root Cause of Check

Determine why a `return` statement does not exist on certain execution paths.

1. Browse the function body for `return` statements.
2. If you find a `return` statement:

   a. See if the `return` statement occurs in a block inside the function.

      For instance, the `return` statement occurs in an `if` block. An execution path that does not enter the `if` block bypasses the `return` statement.

   b. See if you can identify the execution paths that bypass the `return` statement.

      For instance, an `if` block that contains the `return` statement is bypassed for certain function inputs.

   c. If the function is called multiple times in your code, you can identify which function call led to bypassing of the `return` statement. Use the option Sensitivity Context to determine the check color for each function call.

Possible fix: If the return type of the function is incorrect, change it. Otherwise, add a `return` statement on all execution paths. For instance, if only a fraction of branches of an `if-else if-else` condition have a `return` statement, add a `return` statement in the remaining branches. Alternatively, add a `return` statement outside the `if-else if-else` condition.

Review and Fix Illegally Dereferenced Pointer Checks

This topic describes how to systematically review the results of an **Illegally dereferenced pointer** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Illegally dereferenced pointer** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Illegally dereferenced pointer`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2 or “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2.

Step 1: Interpret Check Information

Place your cursor on the dereference operator.

Obtain the following information from the tooltip:

- Whether the pointer can be `NULL`.

  In the following example, `ptr` can be `NULL` when dereferenced.

  ```
  *ptr = 1;
  ```

  Probable cause for 'Illegally dereferenced pointer': Stubbed function 'input'
  Dereference of local pointer 'ptr' (pointer to int 32, size: 32 bits):
  Pointer may be null.
  Points to 4 bytes at unknown offset in buffer of unknown size, so may be outside bounds.
  Pointer may point to dynamically allocated memory.
  Assignment to dereference of local pointer 'ptr' (int 32): 1

  Possible fix: Dereference `ptr` only if it is not `NULL`.

  ```
  if(ptr !=NULL)
      *ptr = 1;
  else
      /* Alternate action */
  ```

- Whether pointer points outside the allocated buffer. A pointer points outside the allocated buffer when the sum of pointer size and offset is greater than buffer size.

  In the following example, the offset size (4096 bytes) together with pointer size (4 bytes) is greater than the buffer size (4096 bytes). If the pointer points to an array:

  - The buffer size is the array size.
  - The offset is the difference between the beginning of the array and the current location of the pointer.

```
*ptr = input();
```

```
dereference of local pointer 'ptr' (pointer to int 32, size: 32 bits):
    pointer is not null
    points to 4 bytes at offset 4096 in buffer of 4096 bytes, so is outside bounds
    may point to variable or field of variable in: {main:arr}
```

Possible fix: Investigate why the pointer points outside the allocated buffer.

- Whether pointer can point outside the allocated buffer because buffer size is unknown.

  In the following example, the buffer size is unknown.

```
val = *ptr;
```

```
Probable cause for 'Non-initialized variable': Stubbed function 'getAddress'
Probable cause for 'Illegally dereferenced pointer': Stubbed function 'getAddress'

dereference of local pointer 'ptr' (pointer to int 32, size: 32 bits):
    pointer is not null (but may not be allocated memory)
    points to 4 bytes at unknown offset in buffer of unknown size, so may be outside bounds
    may point to dynamically allocated memory
dereferenced value (int 32): full-range [-2^31 .. 2^31-1]
```

  Possible fix: Investigate whether the pointer is assigned:

  - The return value of an undefined function.
  - The return value of a dynamic memory allocation function. Sometimes, Polyspace cannot determine the buffer size from the dynamic memory allocation.
  - Another pointer of a different type, for instance, `void*`.

- The probable root cause for illegal pointer dereference, if indicated in the tooltip.

  In the following example, the software identifies a stubbed function, `getAddress`, as probable cause.

```
val = *ptr;
```

```
Probable cause for 'Non-initialized variable': Stubbed function 'getAddress'
Probable cause for 'Illegally dereferenced pointer': Stubbed function 'getAddress'

dereference of local pointer 'ptr' (pointer to int 32, size: 32 bits):
    pointer is not null (but may not be allocated memory)
    points to 4 bytes at unknown offset in buffer of unknown size, so may be outside bounds
    may point to dynamically allocated memory
dereferenced value (int 32): full-range [-2^31 .. 2^31-1]
```

  Possible fix: To avoid the illegally dereferenced pointer, constrain the return value of `getAddress` to provide design information that exists outside your code. For instance, you might specify that

getAddress returns a pointer to a 10-element array. For more information, see “Code Prover Assumptions About Stubbed Functions”.

Step 2: Determine Root Cause of Check

Select the check and note the information on the **Result Details** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction and trace back to the root cause.
- If the **Result Details** pane shows the line number of probable cause for the check, in the Polyspace user interface, right-click the **Source** pane. Select **Go To Line**.
- Otherwise, based on the nature of the error, use one of the following methods to find the root cause. You can perform the following steps in the Polyspace user interface only.

Error	How to Find Root Cause

Pointer can be NULL.	Find an execution path where the pointer is assigned the value NULL or not assigned a definite address.
	
	1. Right-click the pointer and select **Search For All References**.
	2. Find each previous instance where the pointer is assigned an address.
	3. For each instance, on the **Source** pane, place your cursor on the pointer. The tooltip indicates whether the pointer can be NULL.
	
	Possible fix: If the pointer can be NULL, place a check for NULL immediately after the assignment.
	
	`if(ptr==NULL)`
	`  /* Error handling*/`
	`else {`
	`  .`
	`  .`
	`    }`
	
	4. If the pointer is not NULL, see if the assignment occurs only in a branch of a conditional statement. Investigate when that branch does not execute.
	
	Possible fix: Assign a valid address to the pointer in all branches of the conditional statement.

<table>
<tr><th>Error</th><th>How to Find Root Cause</th></tr>
<tr><td>Pointer can point outside bounds allowed by the buffer.</td><td>
1. Find the size of the allocated buffer.
a. On the Search tab, enter the name of the variable that the pointer points to. You already have this name from the tooltip on the check.
b. Search for the variable definition. Typically, this is the first search result.

If the variable is an array, note the array size. If the variable is a structure, search for the structure type name on the Search tab and find the structure definition. Note the size of the structure field that the pointer points to.
2. Find out why the pointer points outside the allocated buffer.
a. Right-click the pointer and select Search For All References.
b. Identify any increment or decrement of the pointer. See if you intended to make the increment or decrement.

Possible fix: Remove unintended pointer arithmetic. To avoid pointer arithmetic that takes a pointer outside the allocated buffer, use a reference pointer to store its initial value. After every arithmetic operation on your pointer, compare it with the reference pointer to see if the difference is outside the allocated buffer.
</td></tr>
</table>

Step 3: Look for Common Causes of Check

Look for common causes of the **Illegally dereferenced pointer** check.

- If you use pointers for moving through an array, see if you can use an array index instead.

  To avoid use of pointer arithmetic in your code, look for violations of MISRA C: 2004 rule 17.4 or MISRA C: 2012 rule 18.4. For more information, see “Check for and Review Coding Standard Violations” on page 16-2.
- See if you use pointers for moving through the fields of a structure.

  Polyspace does not allow the pointer to one field of a structure to point to another field. To allow this behavior, use the option `Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)`.
- See if you are dereferencing a pointer that points to a structure but does not have sufficient memory for all its fields. Such a pointer usually results from type-casting a pointer to a smaller structure.

  Polyspace does not allow such dereference. To allow this behavior, use the option `Allow incomplete or partial allocation of structures (-size-in-bytes)`.
- If an orange check occurs in a function body, see if you are passing arrays of different sizes in different calls to the function.

See if one particular call causes the orange check. For a tutorial, see “Identify Function Call with Run-Time Error” on page 22-44.

- See if you are performing a cast between two pointers of incompatible sizes.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2 or “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2.

For instance, suppose that the pointer receives an address from an undefined function. Then:

1. Polyspace assumes that the function can return `NULL`.

   Therefore, the pointer dereference is orange.
2. Polyspace also assumes a buffer size based on the type of the pointer.

   If you increment the pointer, you exceed the allocated buffer. The pointer dereference that follows the increment is orange.
3. If you know that the function returns a non-`NULL` value or if you know the true allocated buffer, add a comment and justification in your code describing why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Note Before justifying an orange check, consider carefully whether you can improve your coding design.

Review and Fix Incorrect Object Oriented Programming Checks

In this section...
"Step 1: Interpret Check Information" on page 22-22
"Step 2: Determine Root Cause of Check" on page 22-22

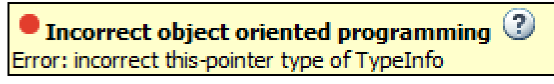
This topic describes how to systematically review the results of an **Incorrect object oriented programming** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Incorrect object oriented programming** check. For a description of the check and code examples, see `Incorrect object oriented programming`.

For the general workflow that applies to all checks, see "Interpret Code Prover Results in Polyspace Desktop User Interface" on page 21-2 or "Interpret Code Prover Results in Polyspace Access Web Interface" on page 26-2.

Step 1: Interpret Check Information

On the **Results List** pane, select the check. The **Result Details** pane displays further information about the check.



You can see:

- The immediate cause of the check. For instance:
  - You dereference a function pointer that has the value `NULL` or points to an invalid member function.

    The member function is invalid if its argument or return type does not match the pointer argument or return type.
  - You call a pure `virtual` member function of a class from the class constructor or destructor.
  - You call a member function using an incorrect `this` pointer.

    To see why the `this` pointer can be incorrect, see `Incorrect object oriented programming`.
- The probable root cause of the check, if indicated.

Step 2: Determine Root Cause of Check

If you cannot determine the root cause based on the check information, use navigation shortcuts in the user interface to navigate to the root cause.

Based on the specific error, use one of the following methods to find the root cause.

Error	How to Find Root Cause
You dereference a function pointer that has the value `NULL`.	Right-click the function pointer and select **Search For All References**. Find the instance where you assign `NULL` to the function pointer.
You dereference a function pointer that points to an invalid member function.	Compare the argument and return types of the function pointer and the member function that it points to. 1 Right-click the function pointer on the **Source** pane and select **Search For All References**. Find the instances where you: • Define the function pointer. • Assign the address of a member function to the function pointer. 2 Find the member function definition. Right-click the member function name on the **Source** pane and select **Go To Definition**.
You call a pure `virtual` member function from a constructor or destructor.	Find the member function declaration and determine whether you intended to declare it as `virtual` or pure `virtual`. Alternatively, determine if you can replace the call to the pure `virtual` function with another operation, for instance, a call to a different member function. 1 Right-click the function name on the **Source** pane and select **Search for *`function_name`* in All Source Files**. 2 Find the function declaration from the search results. A pure `virtual` function has a declaration such as: `virtual void func() = 0;`

You call a member function using an incorrect `this` pointer.	Determine why the `this` pointer is incorrect.
	
	For instance, if a red **Incorrect object oriented programming** check appears on a function call `ptr->func()` and the message indicates that the `this` pointer is incorrect, trace the data flow for `ptr`.
	
	• Right-click the function pointer on the **Source** pane and select **Search For All References**.
	• Browse through all write operations on the pointer. Look for the following issues:
	  • Cast between pointers of unrelated types.
	  • Pointer arithmetic that takes a pointer outside its allocated buffer, for instance, the bounds of an array.
	
	If a red **Incorrect object oriented programming** check appears on a function call `obj.func()`, trace the data flow for `obj`. See if `obj` is not initialized previously.

Review and Fix Invalid C++ Specific Operations Checks

This topic describes how to systematically review the results of an **Invalid C++ specific operations** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Invalid C++ specific operations** check. There are multiple ways to fix a red or orange check. For a description of the check and code examples, see `Invalid C++ specific operations`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see "Interpret Code Prover Results in Polyspace Desktop User Interface" on page 21-2 or "Interpret Code Prover Results in Polyspace Access Web Interface" on page 26-2.

Step 1: Interpret Check Information

On the **Results List** pane, select the check. The **Result Details** pane displays further information about the check.

? **C++ specific checks**
Warning: typeid argument may be incorrect
This check may be an issue related to unbounded input values
If appropriate, applying DRS to stubbed function getShapePtr() in file_typeid.cpp line 53 may remove this orange.

You can see:

- The immediate cause of the check. For instance:
  - The size of an array is not strictly positive.

    For instance, you create an array using the statement `arr = new char [num]`. `num` is possibly zero or negative.

    Possible fix: Use `num` as an array size only if it is positive.
  - The `typeid` operator dereferences a possibly `NULL` pointer.

    Possible fix: Before using the `typeid` operator on a pointer, test the pointer for `NULL`.
  - The `dynamic_cast` operator performs an invalid cast.

    Possible fix: The invalid cast results in a `NULL` return value for pointers and the `std::bad_cast` exception for references. Try to avoid the invalid cast. Otherwise, if the invalid cast is on pointers, make sure that you test the return value of `dynamic_cast` for `NULL` before dereference. If the invalid cast is on references, make sure that you catch the `std::bad_cast` exception in a `try-catch` statement.
- The probable root cause of the check, if indicated.

Step 2: Determine Root Cause of Check

If you cannot determine the root cause based on the check information, use navigation shortcuts in the user interface to navigate to the root cause.

Based on the nature of the error, use one of the following methods to find the root cause.

Error	How to Find Root Cause
An array size is nonpositive.	1 Trace the data flow for the size variable. Follow the same root cause investigation steps as for a **Division by Zero** check. See “Review and Fix Division by Zero Checks” on page 22-7. 2 Identify a point where you can constrain the array size variable to positive values.
The `typeid` operator dereferences a possibly `NULL` pointer.	1 Trace the data flow for the pointer variable. Follow the same root cause investigation steps as for an **Illegally dereferenced pointer** check. See “Review and Fix Illegally Dereferenced Pointer Checks” on page 22-17. 2 Identify a point where you can test the pointer for `NULL`.
The `dynamic_cast` operator performs an invalid cast.	Navigate to the definitions of the classes involved. Determine the inheritance relationship between the classes. 1 On the **Source** pane in the Polyspace user interface, right-click the class name. 2 Select **Go To Definition**.

Step 3: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2 or “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2.

For instance, you obtain the array size variable from a stubbed function `getSize`. Then:

1. Polyspace assumes that the return value of `getSize` is full-range. The range includes nonpositive values.
2. Using the variable as array size in dynamic memory allocation causes orange **Invalid C++ specific operations**.
3. If you know that the variable takes a positive value, add a comment and justification explaining why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Note Before justifying an orange check, consider carefully whether you can improve your coding design.

Review and Fix Invalid Shift Operations Checks

This topic describes how to systematically review the results of an **Invalid shift operations** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Invalid shift operations** check. There are multiple ways to fix the check. For a description of the check and code examples, see `Invalid shift operations`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2 or “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2.

Step 1: Interpret Check Information

Select the red or orange **Invalid shift operations** check. Obtain the following information from the **Result Details** pane:

- The reason for the check being red or orange. Possible reasons:

  - The shift amount can be outside allowed bounds.

    The software also states the allowed range for the shift amount.
  - Left operand of left shift can be negative.

  In the example below, a red error occurs because the shift amount is outside allowed bounds. The allowed range for the shift amount is 0 to 31.

  Invalid shift operations
  Error: left operand of left shift is negative
  This check may be a path-related issue, which is not dependent on input values
  operator << on type int 32
  left: -200
  right: 1

  Possible fix: To avoid the red or orange check, perform the shift operation only if the shift amount is within bounds.

  ```
  if(shiftAmount < (sizeof(int) * 8))
    /* Perform the shift */
  else
    /* Error handling */
  ```
- Probable root cause for the check, if the software provides this information.

Invalid shift operations
Warning: left operand of left shift may be negative
This check may be an issue related to unbounded input values
If appropriate, applying DRS to stubbed function getVal in shf_2.c line 9 may remove this orange.
If appropriate, applying DRS to stubbed function getVal in shf_2.c line 8 may remove this orange.
operator << on type int 32
left: [-65535 .. 65535]
right: 10
result: multiples of 1024 in [0 .. 67107840 (0x3FFFC00)]

In the preceding example, the software identifies a stubbed function, `getVal` as probable cause.

Possible fix: To avoid the orange check, constrain the return value of `getVal`. For instance, specify that `getVal` returns values in a certain range, for example, `0..10`. For more information, see "Code Prover Assumptions About Stubbed Functions".

Step 2: Determine Root Cause of Check

- If the shift amount is outside bounds, trace the data flow for the shift variable. Identify a suitable point where you can constrain the shift variable.

  In the following example, trace the data flow for `shiftAmount`.

  ```
  void func(int val) {
      int shiftAmount = getShiftAmount();
      int res = val >> shiftAmount;
  }
  ```

  You might find that `getShiftAmount` returns full-range of values.

  Possible fix:

  - Perform the shift operation only if `shiftAmount` is between 0 and `(sizeof(int))*8 - 1`.
  - Constrain the return value of `getShiftAmount`, in the body of `getShiftAmount` or through the Polyspace Constraint Specification interface, if you do not have the definition of `getShiftAmount`. For more information, see "Code Prover Assumptions About Stubbed Functions".
- If the left operand of a left shift operation can be negative, trace the data flow for the left operand variable. Identify a suitable point where you can constrain the left operand variable.

  In the following example, trace the data flow for `shiftAmount`.

  ```
  void func(int shiftAmount) {
      int val = getVal();
      int res = val << shiftAmount;
  }
  ```

  You might find that `getVal` returns full-range of values.

  Possible fix:

  - Perform the shift operation only if `val` is positive.
  - Constrain the return value of `getVal`, in the body of `getVal` or through the Polyspace Constraint Specification interface, if you do not have the definition of `getVal`. For more information, see "Code Prover Assumptions About Stubbed Functions".

- If you want Polyspace to allow the operation, use the analysis option **Allow negative operand for left shifts**. See `Allow negative operand for left shifts (-allow-negative-operand-in-shift)`.

To trace the data flow, select the check and note the information on the **Result Details** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Result Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.
- Otherwise:

  1. Find the previous write operation on the variable you want to trace.
  2. At the previous write operation, identify a new variable to trace back.

     Place your cursor on the variables involved in the write operation to see their values. The values help you decide which variable to trace.
  3. Find the previous write operation on the new variable. Continue tracing back in this way until you identify a point to specify your constraint.

  Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. 1. Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2. On the **Search** pane, select the previous instances. • Browse the source code. 1. Double-click the variable on the **Source** pane. All instances of the variable are highlighted. 2. Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1. Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. 2. On this pane, select the previous instances of the variable. Write operations on the variable are indicated with and read operations with .

Variable	How to Find Previous Instances of Variable
Function return value `ret=func();`	**1** Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration. **2** In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 22-77.

Step 3: Look for Common Causes of Check

Look for common causes of the **Invalid Shift Operations** check.

- See if you have specified the right target processor type. The target processor type determines the number of bits allowed for a certain variable type.

  To determine the number of bits allowed:

  1. Navigate to the variable definition. Note the variable type.

     Right-click the variable and select **Go To Definition**, if the option exists.
  2. See the number of bits allowed for the type.

     In the configuration used for your results, select the **Target & Compiler** node. Click the **Edit** button next to the **Target processor type** list.
- For left shifts with a negative operand, see if you intended to perform a right shift instead.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2 or “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2.

For instance, you obtain a variable from an undefined function and perform a left shift on it. Then:

1. Polyspace assumes that the function can return a negative value.
2. The left shift operation can occur on a negative value and therefore there is an orange check on the operation.
3. If you know that the function returns a positive value, add a comment and justification describing why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Review and Fix Invalid Use of Standard Library Routine Checks

This topic describes how to systematically review the results of an **Invalid use of standard library routine** check in Polyspace Code Prover.

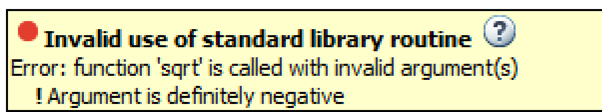
Follow one or more of these steps until you determine a fix for the **Invalid use of standard library routine** check. For a description of the check and code examples, see `Invalid use of standard library routine`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see "Interpret Code Prover Results in Polyspace Desktop User Interface" on page 21-2 or "Interpret Code Prover Results in Polyspace Access Web Interface" on page 26-2.

Step 1: Interpret Check Information

Select the check on the **Results List** pane. View further information about the check on the **Result Details** pane. The check is red or orange because of invalid function arguments.



The cause of a red or orange check depends on the standard library function that you use. The following table shows the possible causes for some of the commonly used functions.

<table>
<tr><th>Function</th><th colspan="2">Cause of Red or Orange Check</th></tr>
<tr><td><code>islower</code>, <code>isdigit</code>, and other character-handling functions in <code>ctype.h</code></td><td colspan="2">The value of the argument can be outside the range allowed for an <code>unsigned char</code> variable.

Note that you can use the macro <code>EOF</code> as argument.</td></tr>
<tr><td rowspan="4">Functions in <code>math.h</code></td><td colspan="2">The software checks for multiple kinds of errors in sequence. The software performs each check only for those execution paths where the previous check passes.

Some examples are given below. For more information and a list of functions, see "Invalid Use of Standard Library Floating Point Routines" on page 22-32.</td></tr>
<tr><td><code>sqrt</code></td><td>The value of the argument can be negative.</td></tr>
<tr><td><code>pow</code></td><td>The first argument can be negative while the second argument is a non-integer.</td></tr>
<tr><td><code>exp</code>, <code>exp2</code>, or the hyperbolic functions</td><td>The argument can be so large that the result exceeds the value allowed for a <code>double</code>.</td></tr>
</table>

Function	Cause of Red or Orange Check	
	log	The argument can be zero or negative.
	asin or acos	The argument can be outside the range [-1,1].
	tan	The argument can have the value HALF_PI.
	acosh	The argument can be less than 1.
	atanh	The argument can be greater than 1 or less than -1.
fprintf, fscanf, and other file handling functions	The file pointer argument can be non-readable. For example, it can be NULL.	
Functions that take string arguments	The string argument can be an invalid string. For example, it does not end with a terminating '\0'.	
memmove or memcpy	The third argument of this function specifies the number of bytes to copy from the second to the first argument. This number can exceed the memory allocated to the first or second argument.	

Step 2: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2 or “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2.

For instance, you obtain a value from an undefined function and perform the `sqrt` operation on it. Then:

1. Polyspace assumes that the function can return a negative value.
2. Therefore, the software produces an orange **Invalid Use of Standard Library Routine** check on the `sqrt` function call.
3. If you know that the function returns a positive value, to avoid the orange, you can specify a constraint on the return value of your function. See “Code Prover Assumptions About Stubbed Functions”. Alternately, add a comment and justification describing why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Invalid Use of Standard Library Floating Point Routines

This topic describes **Invalid use of standard library routine** checks on floating-point routines in Polyspace Code Prover.

Polyspace Code Prover performs the **Invalid Use of Standard Library Routine** check on standard library routines to determine if their arguments are valid. The check works differently for memory routines, floating-point routines or string routines because their arguments can be invalid in different ways. This topic describes how the check works for standard library floating-point routines.

For more information on the check, see `Invalid use of standard library routine`.

What the Check Looks For

The **Invalid Use of Standard Library Routine** check sequentially looks for the following issues in use of floating-point routines.

- Domain error: A domain error occurs if the arguments of the function are invalid. The definition of invalid argument varies based on whether you allow non-finite floats or not. If you allow non-finite floats but:

  - Specify that you must be warned about NaN results, a domain error occurs if the function returns NaN and the arguments themselves are not NaN.
  - Specify that NaN results must be forbidden, a domain error occurs if the function returns NaN or the arguments themselves are NaN.

  For details, see `NaNs (-check-nan)`.

  The check works in almost the same way as the check `Invalid operation on floats`. The **Invalid Use of Standard Library Routine** check works on standard library functions while the **Invalid Operation on Floats** check works on numerical operations involving floating-point variables.
- Overflow error: An overflow error occurs if the result of the function overflows. The definition of overflow varies based on whether you allow non-finite floats and based on the rounding modes you specify. If you allow non-finite floats but specify that you must be warned about infinite results, an overflow error occurs if the function returns infinity and the arguments themselves are not infinity. For details, see `Infinities (-check-infinite)`.

  The check works in the same way as the check `Overflow`. The **Invalid Use of Standard Library Routine** check works on standard library functions while the **Overflow** check works on numerical operations involving floating-point variables.
- Invalid pointer argument: For functions such as `frexp` that take pointer arguments, the verification checks if it is valid to dereference the pointer. For instance, the pointer is not NULL or does not point outside allowed bounds.

The check looks for these errors in sequence.

- If the check finds a definite domain error, it does not look for the overflow error.
- If the check finds a possible domain error, it looks for the overflow error only for the execution paths where the domain error does not occur.

The check for each error itself can consist of multiple conditions, which are also checked in sequence. Each check is performed only for those execution paths where the previous check passes.

Single-Argument Functions Checked

The **Invalid Use of Standard Library Routine** check covers the following routines, their single-precision versions with suffix `f` (if they have one) and their long double versions with suffix `l`. The check works in exactly the same way for C and C++ code.

- `acos`
- `acosh`
- `asin`
- `asinh`
- `atan`
- `atanh`
- `ceil`
- `cos`
- `cosh`
- `exp`
- `exp2`
- `expm1`
- `fabs`
- `floor`
- `log`
- `log10`
- `log1p`
- `logb`
- `round`
- `sin`
- `sinh`
- `sqrt`
- `tan`
- `tanh`
- `trunc`
- `cbrt`

Functions with Multiple Arguments

The **Invalid Use of Standard Library Routine** check covers the following routines, their single-precision versions with suffix `f` (if they have one) and their long double versions with suffix `l`. The check works in exactly the same way for C and C++ code.

- `atan2`
- `fdim`
- `fma`

- `fmax`
- `fmin`
- `fmod`
- `frexp`
- `hypot`
- `ilogb`
- `ldexp`
- `modf`
- `nextafter`
- `nexttoward`
- `pow`
- `remainder`

See Also

Consider non finite floats (-allow-non-finite-floats) | Float rounding mode (-float-rounding-mode)

Review and Fix Non-initialized Local Variable Checks

This topic describes how to systematically review the results of a **Non-initialized local variable** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Non-initialized local variable** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Non-initialized local variable`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2 or “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2.

Step 1: Interpret Check Information

Place your cursor on the variable on which the **Non-initialized local variable** error appears.

```
val++;
```

Probable cause for 'Non-initialized local variable': Stubbed function 'initialize'

Assignment to local variable 'val' (int 32): $[-2^{31}+1 .. 2^{31}-1]$

Local variable 'val' (int 32): full-range $[-2^{31} .. 2^{31}-1]$

Obtain the probable root cause for the variable being non-initialized, if indicated in the tooltip.

In the preceding example, the software identifies a stubbed function, `initialize`, as probable cause.

Possible fix: To avoid the check, you can specify that `initialize` writes to its arguments. For more information, see “Code Prover Assumptions About Stubbed Functions”.

Step 2: Determine Root Cause of Check

You can perform the following steps in the Polyspace user interface only.

1. Search for the variable definition. See if you initialize the variable when you define it.

   Right-click the variable and select **Go To Definition**, if the option exists.
2. If you do not want to initialize the variable during definition, browse through all instances of the variable. Determine if you initialize the variable in any of those instances.

   Do one of the following:

   - On the **Source** pane, double-click the variable.

Previous instances of the variable are highlighted. Scroll up to find them.

- On the **Source** pane, right-click the variable. Select **Search For All References**.

  Select the previous instances on the **Search** pane.

Possible fix: If you do not initialize the variable, identify an instance where you can initialize it.

3 If you find an instance where you initialize the variable, determine if you perform the initialization in the scope where the **Non-initialized local variable** error appears.

For instance, you initialize the variable only in some branches of an `if ... elseif ... else` statement. If you use the variable outside the statement, the variable can be non-initialized.

Possible fix:

- Perform the initialization in the same scope where you use it.

  In the preceding example, perform the initialization outside the `if ... elseif ... else` statement.
- Perform the initialization in a block with smaller scope but make sure that the block always executes.

  In the preceding example, perform the initialization in all branches of the `if ... elseif ... else` statement. Make sure that one branch of the statement always executes.

Step 3: Look for Common Causes of Check

Look for common causes of the **Non-initialized local variable** check.

- See if you pass the variable to another function by reference or pointers before using it. Determine if you initialize the variable in the function body.

  To navigate to the function body, right-click the function and select **Go To Definition**, if the option exists.
- Determine if you initialize the variable in code that is not reachable.

  For instance, you initialize the variable in code that follows a `break` or `return` statement.

  Possible fix: Investigate the unreachable code. For more information, see "Review and Fix Unreachable Code Checks" on page 22-68.
- Determine if you initialize the variable in code that can be bypassed during execution.

  For instance, you initialize the variable in a loop inside a function. However, for certain function arguments, the loop does not execute.

  Possible fix:

  - Initialize the variable during declaration.
  - Investigate when the code can be bypassed. Determine if you can avoid bypassing of the code.
- If the variable is an array, determine if you initialize all elements of the array.
- If the variable is a structured variable, determine if you initialize all fields of the structure.

  If you do not initialize a certain field of the structure, see if the field is unused.

Possible fix: Initialize a field of the structure if you use the field in your code.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" on page 23-2 or "Address Results in Polyspace Access Through Bug Fixes or Justifications" on page 27-2.

For instance, you pass a variable to a function by pointer or reference. You intend to initialize the variable in the function body, but you do not provide the function body during verification. Then:

- Polyspace assumes that the function might not initialize the variable.
- If you use the variable following the function call, Polyspace considers that the variable can be non-initialized. It produces an orange **Non-initialized local variable** check on the variable.

For more information, see "Code Prover Analysis Assumptions".

Note Before justifying an orange check, consider carefully whether you can improve your coding design.

Disabling This Check

You can disable this check. If you disable this check, Polyspace assumes that at declaration, variables have full-range of values allowed by their type. For more information, see `Disable checks for non-initialization (-disable-initialization-checks)`.

Review and Fix Non-initialized Pointer Checks

This topic describes how to systematically review the results of a **Non-initialized pointer** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Non-initialized pointer** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Non-initialized pointer`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2 or “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2.

Step 1: Interpret Check Information

Select the check on the **Results List** pane. On the **Result Details** pane, obtain further information about the check.

Non-initialized pointer
Warning: pointer may be non-initialized
Dereferenced value (pointer to int 8, size: 8 bits):
Pointer is not null.
Points to 1 bytes at offset [1 .. 9] in buffer of 20 bytes, so is within bounds (if memory is allocated).
Pointer may point to variable or field of variable:
'arr', local to function 'main'.

Step 2: Determine Root Cause of Check

Right-click the pointer variable and select **Go To Definition**. Initialize the variable when you define it. If you do not want to initialize during definition, identify a suitable point to initialize the variable before you read it.

For orange checks, determine why the pointer is non-initialized on certain execution paths.

1. Find previous instances where write operations are performed on the pointer.
2. For each write operation, determine if the operation occurs:

   - Before the read operation containing the orange **Non-initialized pointer** check.

     Possible fix: If the write operation occurs after the read operation, see if you intended to perform the operations in reverse order.
   - In an unreachable code block.

     Possible fix: Investigate why the code block is unreachable. See “Review and Fix Unreachable Code Checks” on page 22-68.
   - In a code block that is not reached on certain execution paths. For example, the operation occurs in an `if` block in a function. The `if` block is not entered for certain function inputs.

Possible fix: Perform a write operation on all the execution paths. In the preceding example, perform the write operation in all branches of the `if ... elseif ... else` statement.

Depending on the nature of the variable, use the appropriate method to find previous operations on the variable. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Operations on Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 On the **Source** pane, double-click the variable. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. The current instance of the variable is shown on the **Variable Access** pane. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with . Read operations are indicated with .

Step 3: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" on page 23-2 or "Address Results in Polyspace Access Through Bug Fixes or Justifications" on page 27-2.

Disabling This Check

You can disable the check in two ways:

- You can disable the check only for non-local pointers. Polyspace considers global pointer variables to be initialized to `NULL` according to ANSI C standards. For more information, see Ignore default initialization of global variables.
- You can disable the check completely along with other initialization checks. If you disable this check, Polyspace assumes that at declaration, pointers can be `NULL` or point to memory blocks at an unknown offset. For more information, see `Disable checks for non-initialization (-disable-initialization-checks)`.

Review and Fix Non-initialized Variable Checks

This topic describes how to systematically review the results of a **Non-initialized variable** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Non-initialized variable** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Non-initialized variable`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2 or “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2.

Step 1: Interpret Check Information

On the **Results List** pane, select the check. On the **Result Details** pane, obtain further information about the check.

Non-initialized variable
Warning: variable may be non-initialized (type: int 32)
This check may be a path-related issue, which is not dependent on input values
Global variable 'globVar' (int 32): 0

Obtain the following information:

- Probable cause of check, if described on the **Result Details** pane.

  In the preceding example, there is an orange **Non-initialized variable** check on the global variable `globVar`.

  The software detects that the check is potentially a path-related issue. Therefore, the global variable can be non-initialized only on certain execution paths. For example, you initialized the global variable in an `if` block, but did not initialize it in the corresponding `else` block.

  Possible fix: Determine along which paths the global variables can be non-initialized.
- Value of global variable, if initialized.

  In the preceding example, when initialized, the global variable `globVar` has value 0.

Step 2: Determine Root Cause of Check

You can perform the following steps in the Polyspace user interface only.

Right-click the variable and select **Go To Definition**. Initialize the variable when you define it. If you do not want to initialize during definition, identify a suitable point to initialize the variable before you read it.

If the check is orange, determine why the variable is non-initialized on certain execution paths.

1. Right-click the variable. Select **Show In Variable Access View**.
2. On the **Variable Access** pane, select each write operation on the variable.

   Write operations are indicated with ◀ and read operations with ▶.
3. Determine if the write operation occurs:

   - Before the read operation containing the orange **Non-initialized variable** check.

     Possible fix: If the write operation occurs after the read operation, see if you intended to perform the operations in reverse order.
   - In an unreachable code block.

     Possible fix: Investigate why the code block is unreachable. See “Review and Fix Unreachable Code Checks” on page 22-68.
   - In a code block that is not reached on certain execution paths. For example, the operation occurs in an `if` block in a function. The `if` block is not entered for certain function inputs.

     Possible fix: Perform a write operation on all the execution paths. In the preceding example, perform the write operation in all branches of the `if ... elseif ... else` statement.

Step 3: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2 or “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2.

Disabling This Check

You can disable this check in two ways:

- You can specify that global variables must be considered as initialized. Polyspace considers global variables to be initialized according to ANSI C standards. The default values are:

  - 0 for `int`
  - 0 for `char`
  - 0.0 for `float`

  For more information, see Ignore default initialization of global variables.
- You can disable the check along with other initialization checks. If you disable this check, Polyspace assumes that at declaration, variables have the full range of values allowed by their type. For more information, see `Disable checks for non-initialization (-disable-initialization-checks)`.

Review and Fix Non-Terminating Call Checks

This topic describes how to systematically review the results of a **Non-terminating call** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Non-terminating call** check. There are multiple ways to fix the check. For a description of the check and code examples, see `Non-terminating call`.

For the general workflow on reviewing checks, see “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2 or “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2.

A red **Non-terminating call** check on a function call indicates one of the following:

- An operation in the function body failed for that particular call. Because there are other calls to the same function that do not cause a failure, the operation failure typically appears as an orange check in the function body.
- The function does not return to its calling context for other reasons. For example, a loop in the function body does not terminate.

Step 1: Determine Root Cause of Check

Determine the root cause of the check in the function body. You can perform the following steps in the Polyspace user interface only.

1. Navigate to the function definition.

   Right-click the function call containing the red check. Select **Go To Definition**, if the option exists.
2. In the function body, determine if there is a loop where the termination condition is never satisfied.

   Possible fix: Change your code or the function arguments so that the termination condition is satisfied.
3. Otherwise, in the function body, identify which orange check caused the red **Non-terminating call** check on the function call.

   If you cannot find the orange check by inspection, rerun verification using the analysis option **Sensitivity context**. Provide the function name as option argument. The software marks the orange check causing the red **Non-terminating call** check as a dark orange check.

   For more information, see `Sensitivity context (-context-sensitivity)`.

   For a tutorial on using the option, see “Identify Function Call with Run-Time Error” on page 22-44.

   Possible fix: Investigate the cause of the orange check. Change your code or the function arguments to avoid the orange check.

Step 2: Look for Common Causes of Check

To trace a **Non-terminating call** check on a function call to an orange check in the function body, try the following:

- If the function call contains arguments, in the function body, search for all instances of the function parameters. See if you can find an orange check related to the parameters. Because other calls to the same function do not cause an operation failure, it is likely that the failure is related to the function parameter values in the red call.

  In the following example, in the body of `func`, search for all instances of `arg1` and `arg2`. Right-click the variable name and select **Search For All References**.

  ```
  void func(int arg1, double arg2) {
    .
    .
  }

  void main() {
    int valInt1,valInt2;
    double valDouble1, valDouble2;
    .
    .
    func(valInt1, valDouble1);
    func(valInt2, valDouble2);
  }
  ```

  Because `valInt1` and `valDouble1` do not cause an operation failure in `func`, the failure might be due to `valInt2` or `valDouble2`. Because `valInt2` and `valDouble2` are copied to `arg1` and `arg2`, the orange check must occur in an operation related to `arg1` or `arg2`.

- If the function call does not contain arguments, identify what is different between various calls to the function. See if you can relate the source of this difference to an orange check in the function body.

  For instance, if the function reads a global variable, different calls to the function can operate on different values of the global variable. Determine if the function body contains an orange check related to the global variable.

Identify Function Call with Run-Time Error

This tutorial shows how to identify the function call that causes a run-time error in the function body.

If a function contains two different colors on the same operation for two different calls, the software combines the call contexts and displays an orange check on the operation. For example, when some function calls cause a red or orange check on an operation in the function body and other calls cause a green check, in your verification results, the operation is orange.

You have to distinguish orange checks that arise from combination of call contexts because an orange check can arise from other causes. Using the option Sensitivity context, make this distinction by storing individual call contexts for a function.

In this tutorial, a function is called twice. You identify which function call causes a run-time error in the function body.

1. Run analysis on this code and open the results.

   ```
   void func(int arg) {
       int loc_var = 0;
       loc_var = 1/arg;
   }

   void main(void) {
       int num = 1;
       func(num + 10);
       func(num - 1);
   }
   ```

   A red **Non-terminating call** check appears on the second call to `func`. In the body of `func`, there is an orange **Division by zero** check on the `/` operation.
2. Specify that you want to store individual call context information for the function `func`.

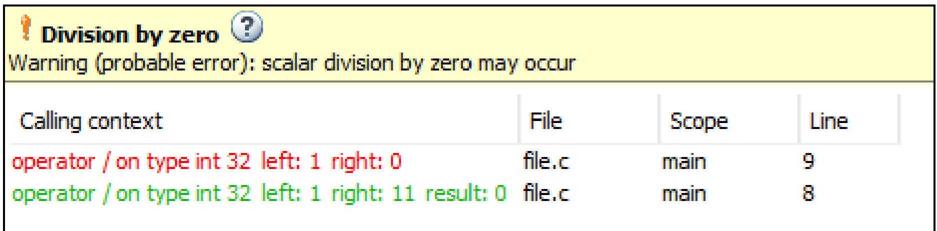
   a. In your project configuration, select the **Precision** node.

   b. Select `custom` for **Sensitivity context**.

   c. Click ✚ to add a new field. Enter `func`.
3. Run verification and open the results.

   An orange **Division by zero** check still appears in the body of `func`. However, this orange check is marked on the **Results List** pane as a dark orange check and is denoted by a ! mark. Instead of a red **Non-terminating call** check, a dashed, red line appears on the second call to `func`.
4. Select the orange check.

   The **Result Details** pane shows the call contexts for the check. You can see that one call produces a red check on the `/` operation and the other call produces a green check. You can click each call to navigate to it in your source code.



See Also

`Non-terminating call`

Related Examples

- “Review and Fix Non-Terminating Call Checks” on page 22-42

More About

- “Orange Checks in Polyspace Code Prover” on page 33-2

Review and Fix Non-Terminating Loop Checks

This topic describes how to systematically review the results of a **Non-terminating loop** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Non-terminating loop** check. There are multiple ways to fix the check. For a description of the check and code examples, see `Non-terminating loop`.

For the general workflow on reviewing checks, see “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2 or “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2.

Step 1: Interpret Check Information

Place your cursor on the loop keyword such as `for` or `while`.

Obtain the following information from the tooltip:

- Whether the loop is infinite or contains a run-time error.

  In the following example, it is likely that the loop is infinite.

```
while(i<10) {
```
loop may be infinite

- If the loop contains a run-time error, the number of loop iterations. Because Polyspace considers that execution stops when a run-time error occurs, from this number, you can determine which loop iteration contains the error.

  In the following example, it is likely that the loop contains a run-time error. The error is likely to occur on the 31st loop iteration.

```
while(i<1024) {
```
loop may fail due to a run-time error (maximum number of iterations: 31)

Step 2: Determine Root Cause of Check

- If the loop is infinite, determine why the loop-termination condition is never satisfied.

  If you deliberately have an infinite loop in your code, such as for cyclic applications, you can add a comment and justification in your result or code. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2 or “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2.
- If the loop contains a run-time error, identify the error that causes the **Non-terminating loop** check. Fix the error.

  In the loop body, the run-time error typically occurs as an orange check of another type on an operation. The check is orange and not red because the operation typically passes the check in the first few loop iterations and fails only in a later iteration. However, because the failure occurs every time the loop runs, the **Non-terminating loop** check on the loop keyword is red.

For loops with few iterations, you can navigate directly from the loop keyword to the operation causing the run-time error.

- To find the source of error, on the **Source** pane, select the red loop keyword. The **Result Details** pane shows the full history leading to the operation that causes the run-time error.
- Navigate to the source of error in the loop body. Right-click the loop keyword and select **Go to Cause** if the option exists.

```
1  int a[10];
2
3  void foo(int x){
4      for (int i=0; i<=x+5; i++){
5      a[i]=i;
6      }
7  }
8
9  void func(){
10
11
12   int x, i;
13   x = 0;
14   for (i = 0; i <= 10; i++) {
15     a[i+1]=0;
16     foo(i);
17
18   }
19 }
20
```

For a tutorial, see "Identify Loop Operation with Run-Time Error" on page 22-49.

Step 3: Look for Common Causes of Check

- If the loop is infinite:
  - Check your loop-termination condition.
  - Inside the loop body, see if you change at least one of the variables involved in the loop-termination condition.

    For instance, if the loop-termination condition is `while (count1 + count2 < count3)`, see if you are changing at least one of `count1`, `count2`, or `count3` in the loop.
  - If you are changing the variables involved in the loop-termination condition, see if you are changing them in the right direction.

    For instance, if the loop termination condition is `while(i<10)` and you decrement `i` in the loop, the loop does not terminate. You must increment `i`.
- If the loop contains a run-time error:

- If the loop control variable is an array index, see if you have an orange **Out of bounds array index** error in the loop body.
- If the loop control variable is passed to a function, see if you can relate the red **Non-terminating loop** error to an orange error in the function body.

Identify Loop Operation with Run-Time Error

This tutorial shows how to interpret Polyspace Code Prover results that highlight a run-time error inside a loop.

If an error occurs in a loop, the error shows in the analysis results as a red **Non-terminating loop** check on the loop keyword (`for`, `while`, and so on).

```
for (i = 0; i <= 10; i++)
```

The operation causing the error shows as an orange check in the loop. To distinguish this orange check from other orange checks in the loop, navigate directly from the red loop keyword to the operation responsible for the run-time error.

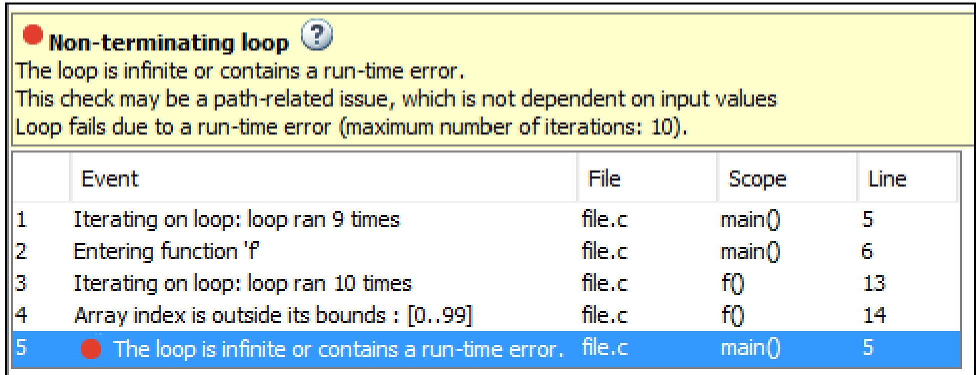
In this tutorial, a function is called in a loop. The function body contains another loop, which has an operation causing a run-time error. You trace from the first loop to the operation causing the run-time error.

1. Run verification on this code and open the results:

   ```
   int a[100];

   int f (int i);

   void main() {
     int i,x=0;
     for (i = 0; i <= 10; i++) {
       x += f(i);
     }
   }

   int f (int i) {
     int j, x;
     x = 0;
     for (j = 0; j <= 10; j++) {
       x += a[10 + (i * j)];
     }
     return x;
   }
   ```
2. Select the red **Non-terminating loop** result. The **Source** pane highlights the `for` loop in `main`.
3. Trace from the `for` loop in `main` to the operation causing the error. The operation is `x+= a[10 + (i*j)]`. An **Out of bounds array index** error occurs when `i` is 9 and `j` is 10. The error shows in orange on the `[` operator.

   To trace from the red `for` keyword to the orange array access operation:

   - Navigate directly to the operation. Right-click the `for` keyword and select **Go to Cause**.
   - See the full history from the `for` keyword to the array access operation. Select the red `for` keyword. The **Result Details** pane shows the history.



You can read the event history in sequence. The outer loop loop runs nine times without error. On the tenth iteration (`i=9`), the loop enters the function `f`. Inside `f`, the inner loop runs ten times without error. On the eleventh iteration (`j=10`), the array access causes an error.

You can use this information to determine how to fix the run-time error on the array access operation.

Note You can navigate directly to the root cause of an error for loops with only a small number of iterations.

See Also

`Non-terminating loop`

Related Examples

- “Review and Fix Non-Terminating Loop Checks” on page 22-46

More About

- “Orange Checks in Polyspace Code Prover” on page 33-2

Review and Fix Null This-pointer Calling Method Checks

In this section...

"Step 1: Interpret Check Information" on page 22-51
"Step 2: Determine Root Cause of Check" on page 22-51

This topic describes how to systematically review the results of a **Null this-pointer calling method** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Null this-pointer calling method** check. For a description of the check and code examples, see `Null this-pointer calling method`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see "Interpret Code Prover Results in Polyspace Desktop User Interface" on page 21-2 or "Interpret Code Prover Results in Polyspace Access Web Interface" on page 26-2.

Step 1: Interpret Check Information

Select the check on the **Results List** pane. The **Result Details** pane displays further information about the check.

Non-null this-pointer in method
Warning: this-pointer of addNewClient may be null
This check may be an issue related to unbounded input values
If appropriate, applying DRS to stubbed function returnPointer() in nnt.cpp line 16 may remove this orange.

You can see:

- The immediate cause of the check.

  In this example, the pointer used to call a method `addNewClient` can be `NULL`.
- The probable root cause of the check, if indicated.

  In this example, the check can be related to a stubbed function `returnPointer`.

Step 2: Determine Root Cause of Check

Find an execution path where the pointer is either assigned the value `NULL` or assigned values from an undefined function or unknown function inputs. In the latter case, the software assumes that the pointer can be `NULL`.

Select the check on the **Results List** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction and trace back to the root cause.

- If the **Result Details** pane shows the line number of probable cause for the check, in the Polyspace user interface, right-click the **Source** pane. Select **Go To Line**.
- If the **Result Details** pane does not lead you to the root cause, using the **Source** pane in the Polyspace user interface, find how the pointer used to call the method can be `NULL`.

  1. Right-click the pointer and select **Search For All References**.
  2. Find each previous instance where the pointer is assigned an address.
  3. For each instance, on the **Source** pane, place your cursor on the pointer. The tooltip indicates whether the pointer can be `NULL`.

     Possible fix: If the pointer can be `NULL`, place a check for `NULL` immediately after the assignment.

     ```
     if(ptr==NULL)
       /* Error handling*/
     else {
       .
       .
         }
     ```
  4. If the pointer is not `NULL`, see if the assignment occurs only in a branch of a conditional statement. Investigate when that branch does not execute.

     Possible fix: Assign a valid address to the pointer in all branches of the conditional statement.

Review and Fix Out of Bounds Array Index Checks

This topic describes how to systematically review the results of an **Out of bounds array index** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Out of bounds array index** check. There are multiple ways to fix the check. For a description of the check and code examples, see `Out of bounds array index`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2 or “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2.

Step 1: Interpret Check Information

Place your cursor on the `[` symbol.

```
val = arr[i];
```

array size: 10
array index value: [0 .. 10]
Element of global array (int 32): full-range [-2^{31} .. $2^{31}-1$]

Obtain the following information from the tooltip:

- Array size. The allowed range for array index is 0 to (array size - 1).
- Actual range for array index

In the preceding example, the array size is 10. Therefore, the allowed range for the array index is 0 to 9. However, the actual range is 0 to 10.

Possible fix: To avoid the out of bounds array index, access the array only if the index is between 0 and (array size - 1).

```
#define SIZE 100
int arr[SIZE];
.
.
if(i<SIZE)
    val = arr[i]
else
    /*Error handling */
```

Step 2: Determine Root Cause of Check

If you want to know or change the array size, right-click the array variable and select **Go To Definition**, if the option exists. Otherwise, trace the data flow starting from the array index variable. Identify a point where you can constrain the index variable.

To trace the data flow, select the check, and note the information on the **Result Details** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Result Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.
- Otherwise:

1. Find previous instances of the array index variable.
2. Browse through those instances. Find the instance where you constrain the array index variable to (array size - 1).

   Possible fix: If you do not find an instance where you constrain the index variable, specify a constraint for the variable. For example:

   ```
   if(index<SIZE)
       read(array[index]);
   ```
3. Determine if the constraint applies to the instance where the **Out of bounds array index** error occurs.

   For example, you can constrain the index variable in a `for` loop using `for(index=0; index<SIZE; index++)`. However, you access the array outside the loop where the constraint does not apply.

   Possible fix: Investigate why the constraint does not apply. See if you have to constrain the index variable again.
4. If the index variable is obtained from another variable, trace the data flow for the second variable. Determine if you have constrained that second variable to (array size - 1).

Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. **1** Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. **2** On the **Search** pane, select the previous instances. • Browse the source code. **1** Double-click the variable on the **Source** pane. All instances of the variable are highlighted. **2** Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	**1** Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. **2** On this pane, select the previous instances of the variable. Write operations on the variable are indicated with and read operations with .
Function return value `ret=func();`	**1** Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration. **2** In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 22-77.

Step 3: Look for Common Causes of Check

Look for common causes of the **Out of bounds array index** check.

- See if you are starting the array index variable from 0.
- In the condition that constrains your array index variable, see if you use <= when you intended to use <.
- If a check occurs in or immediately after a `for` or `while` loop, determine if you have to reduce the number of runs of the loop.
- If you use the `sizeof` function to constrain your array, see if you are dividing `sizeof(array)` by `sizeof(array[0])` to obtain the array size.

  `sizeof(array)` returns the array size in bytes.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2 or “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2.

For instance, you constrain the array index using a function whose definition you do not provide. Then:

1. Polyspace cannot determine that the array index variable is constrained.
2. When you use this variable as array index, an **Out of bounds array index** error can occur.
3. If you know that the variable is constrained to the array size, add a comment and justification describing why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Note Before justifying an orange check, consider carefully whether you can improve your coding design.

For instance, constraining a global variable in one function and using it as array index in a second function causes vulnerabilities in your code. If a new function is called between the previous two functions and modifies your global variable, the constraint no longer applies.

Review and Fix Overflow Checks

This topic describes how to systematically review the results of an **Overflow** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Overflow** check. There are multiple ways to fix the check. For a description of the check and code examples, see `Overflow`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2 or “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2.

Step 1: Interpret Check Information

Place your cursor on the operation that overflows.

```
return(val*2);
```

Probable cause for 'Overflow': Stubbed function 'getVal'

operator * on type int 32

left: full-range [-2^{31} .. 2^{31}-1]

right: 2

result: even values in [-2^{31} .. 2147483646 (0x7FFFFFFE)]

(result is truncated)

Obtain the following information from the tooltip:

- The operand variable you can constrain to avoid the overflow.

  In the preceding example, the left operand, `val`, has full range of values.

  Possible fix: To avoid the overflow, perform the multiplication only if `val` is in a certain range.

  ```
  if(val < INT_MAX/2)
      return(val*2);
  else
      /* Alternate action */
  ```
- The probable root cause for overflow, if indicated in the tooltip.

  In the preceding example, the software identifies a stubbed function, `getVal`, as probable cause.

  Possible fix: To avoid the overflow, constrain the return value of `getVal`. For instance, specify that `getVal` returns values in a certain range, for example, `1..10`. For more information, see “Code Prover Assumptions About Stubbed Functions”.

Step 2: Determine Root Cause of Check

Trace the data flow starting from the operand variable that you want to constrain. Identify a suitable point to specify your constraint.

In the following example, trace the data flow starting from `tempVal`.

```
val = func();
.
.
tempVal = val;
.
.
tempVal++ ;
```

In this example, you might find that:

1. `tempVal` obtains a full-range of values from `val`.

   Possible fix: Assign `val` to `tempVal` only if `val` is less than a certain value.
2. `val` obtains a full-range of values from `func`.

   Possible fix: Constrain the return value of `func`, either in the body of `func` or through the Polyspace Constraint Specification interface, if `func` is stubbed. For more information, see “Code Prover Assumptions About Stubbed Functions”.

To trace the data flow, select the check and note the information on the **Result Details** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Result Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.
- Otherwise:

  1. Find the previous write operation on the operand variable.

     Example: The previous write operation on `tempVal` is `tempVal=val`.
  2. At the previous write operation, identify a new variable to trace back.

     Place your cursor on the variables involved in the write operation to see their values. The values help you decide which variable to trace.

     Example: At `tempVal=val`, you find that `val` has a full-range of values. Therefore, you trace `val`.
  3. Find the previous write operation on the new variable. Continue tracing back in this way until you identify a point to specify your constraint.

     Example: The previous write operation on `val` is `val=func()`. You can choose to specify your constraint on the return value of `func`.

Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 Double-click the variable on the **Source** pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with ◀ and read operations with ▶.
Function return value `ret=func();`	1 Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration. 2 In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

You can also determine if variables in any operation are related from some previous operation. See "Find Relations Between Variables in Code" on page 22-77.

Tip To distinguish between integer and float overflows, use the **Detail** column on the **Results List** pane. Click the column header so that integer and float overflows are grouped together. If you do not see the **Detail** column, right-click any other column header and enable this column.

Step 3: Look for Common Causes of Check

If the operation causing the overflow occurs in a loop or in the body of a recursive function:

- See if you can reduce the number of loop runs or recursions.
- See if you can place an exit condition in the loop or recursive function before the operation overflows.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2 or “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2.

For instance, you are using a volatile variable in your code. Then:

1. Polyspace assumes that the volatile variable is full-range at every step in the code.
2. An operation using that variable can overflow and is therefore orange.
3. If you know that the variable takes a smaller range of values, add a comment and justification in your code describing why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Note Before justifying an orange check, consider carefully whether you can improve your coding design.

Detect Overflows in Buffer Size Computation

If you are computing the size of a buffer from `unsigned` integers, for the **Overflow mode for unsigned integer** option, instead of the default value `allow`, use `forbid`. Using this option helps you detect an overflow at the buffer computation stage. Otherwise, you might see an error later due to insufficient buffer. This option is available on the **Check Behavior** node under **Code Prover Verification** in the **Configuration** pane.

For this example, save the following C code in a file `display.c`:

```
#include <stdlib.h>
#include <stdio.h>

int get_value(void);

void display(unsigned int num_items) {
 int *array;
 array = (int *) malloc(num_items * sizeof(int)); // overflow error
  if (array) {
    for (unsigned int ctr = 0; ctr < num_items; ctr++)     {
      array[ctr] = get_value();
    }
    for (unsigned int ctr = 0; ctr < num_items; ctr++)     {
      printf("Value is %d.\n", ctr, array[ctr]);
    }
    free(array);
  }
}

void main() {
  display(33000);
}
```

1. Create a Polyspace project and add `display.c` to the project.
2. On the **Configuration** pane, select the following options:

   - **Target & Compiler**: From the **Target processor type** drop-down list, select a type with 16-bit `int` such as `c167`.
   - **Check Behavior**: From the **Overflow mode for unsigned integer** drop-down list, select `allow`.
3. Run the verification and open the results.

   Polyspace detects an orange **Illegally dereferenced pointer** error on the line `array[ctr] = get_value()` and a red **Non-terminating loop** error on the `for` loop.

   This error follows from an earlier error. For a 16-bit `int`, there is an overflow on the computation `num_items * sizeof(int)`. Polyspace does not detect the overflow because it occurs in computation with `unsigned` integers. Instead Polyspace wraps the result of the computation causing the **Illegally dereferenced pointer** error later.
4. From the **Overflow mode for unsigned integer** drop-down list, select `forbid`.
5. Polyspace detects a red **Overflow** error in the computation `num_items * sizeof(int)`.

See Also

Polyspace Analysis Options
`Overflow mode for unsigned integer (-unsigned-integer-overflows)`

Polyspace Results
`Overflow` | `Illegally dereferenced pointer`

Review and Fix Return Value Not Initialized Checks

This topic describes how to systematically review the results of a **Return value not initialized** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Return value not initialized** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Return value not initialized`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2 or “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2.

Step 1: Interpret Check Information

Select the check on the **Results List** pane. On the **Result Details** pane, view further information about the check.

? Initialized return value
Warning: function may return a non-initialized value
This check may be a path-related issue, which is not dependent on input values
If appropriate, applying DRS to stubbed function inputRep in file.c line 6 may remove this orange.
Returned value of reply (int 32): full-range [-2^{31} .. $2^{31}-1$]

View the probable cause of check, if mentioned on the **Result Details** pane.

In the preceding example, the software identifies a stubbed function, `inputRep`, as probable cause.

Possible fix: To avoid the check, constrain the argument or return value of `inputRep`. For instance, specify that `inputRep` returns values in a certain range, for example, `1..10`. For more information, see “Code Prover Assumptions About Stubbed Functions”.

Step 2: Determine Root Cause of Check

Determine the root cause of the check in the function body. You can perform the following steps in the Polyspace user interface only.

1. Navigate to the function definition.

   Right-click the function call containing the check. Select **Go To Definition**, if the option exists.
2. In the function body, check if a `return` statement occurs before the closing brace of the function.
3. If a `return` statement does not exist:

   a. On the **Search** pane, search for the word `return`, or manually scroll through the function body and look for `return` statements.
   b. For each `return` statement, determine if the statement appears in a scope smaller than function scope.

For instance, a `return` statement occurs only in one branch of an `if-else` statement.

Possible fix: See if you can place the `return` statement at the end of the function body. For instance, replace the following code

```
int func(int ch) {
    switch(ch) {
        case 1: return 1;
        break;
        case 2: return 2;
        break;
    }
}
```

with

```
int func(int ch) {
    int temp;
    switch(ch) {
        case 1: temp = 1;
        break;
        case 2: temp = 2;
        break;
    }
    return temp;
}
```

For information on how to enforce this practice, see `Number of Return Statements`.

Step 3: Look for Common Causes of Check

Look for common causes of the **Return value not initialized** check.

- See if the `return` statements appear in `if-else`, `for` or `while` blocks. Identify conditions when the function does not enter the block.

  For instance, the function might not enter a `while` block for certain function inputs.

  Possible fix:

  - See if you can place the `return` statement at the end of the function body.
  - Otherwise, determine how to avoid the condition when the function does not enter the block.

    For instance, if a function does not enter a block for certain inputs, see if you must pass different inputs.
- See if you have code constructs such as `goto` that interrupt the normal control flow. See if there are conditions when `return` statements in your function cannot be reached because of these code constructs.

  Possible fix:

  - Avoid `goto` statements. For information on how to enforce this practice, see `Number of Goto Statements`.
  - Otherwise, determine how to avoid the condition when `return` statements in your function cannot be reached.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2 or “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2.

For instance, you have a `return` statement in branches of an `if-elseif` block but you do not have the final `else` block. The condition you are testing in the `if-elseif` blocks involve variables obtained from an undefined function. Then:

1. Polyspace assumes that for certain values of those variables, none of the `if-elseif` blocks are entered.
2. Therefore, it is possible that the `return` statements are not reached.
3. If you know that those values of the variables do not occur, add a comment and justification describing why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Disabling This Check

You can disable this check. If you disable this check, Polyspace assumes the following about a function return value if the function is missing `return` statements:

- If the return value is a non-pointer variable, it has full-range of values allowed by its type.
- If the return value is a pointer, it can be `NULL`-valued or point to a memory block at an unknown offset.

For more information, see `Disable checks for non-initialization (-disable-initialization-checks)`.

Review and Fix Uncaught Exception Checks

This topic describes how to systematically review the results of an **Uncaught exception** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Uncaught exception** check. For a description of the check and code examples, see `Uncaught exception`.

Step 1: Interpret Check Information

Select the check on the **Results List** pane. On the **Result Details** pane, view further information about the check.

The message for a red or orange **Uncaught exception** check typically states one of these reasons.

Message	What This Means
Unhandled exception propagates to main or entry-point function.	An exception is thrown and not handled in a `catch` block. The exception escapes to the `main`.
Call to *`typeName`* throws during "catch" parameter construction.	Creating the `catch` parameter invokes a constructor. The constructor throws an exception.
Throw during destructor or delete.	A destructor throws an exception.

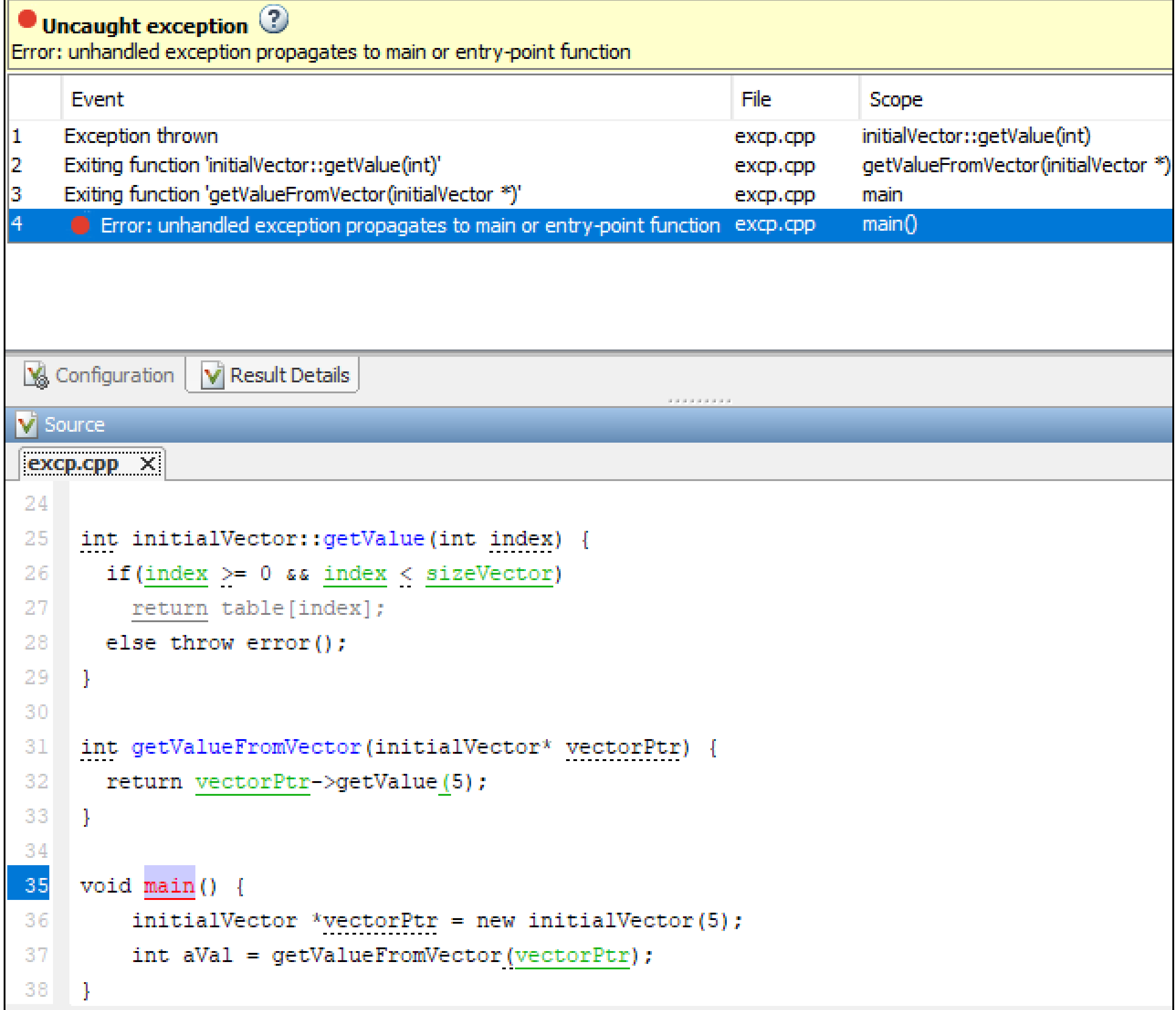
Step 2: Determine Root Cause of Check

The most common root cause is that an exception propagates up the function call hierarchy from its origin to the `main` function.

In the event traceback associated with the check, you see the origin of the exception and one path up the function call tree to the `main` or another entry-point function. Click each event to navigate to the corresponding point in the source code.

In this example, the exception is thrown in the method `initialVector::getValue` which is called from the `main` in this sequence:

- `main`
- `getValueFromVector`
- `initialVector::getValue`



The event list shows these points in the code:

1. The statement that throws an exception.
2. The return from the function where the exception is thrown, in this case, the `initialVector::getValue` method.
3. The return from the next function that the exception propagates to, in this case, the `getValueFromVector` method.
4. The `main` function.

Using this event list, you can trace how the exception escapes and place a `try-catch` block to handle the exception. For instance, you can place the call:

```
return vectorPtr->getValue(5)
```

in a `try-catch` block. In the `catch` block, you can catch an exception of type `error`.

Review and Fix Unreachable Code Checks

This topic describes how to systematically review the results of an **Unreachable code** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **Unreachable code** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Unreachable code`.

If you determine that the check represents defensive code, add a comment and justification in your result or code explaining why you did not change your code. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2 or “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2.

Step 1: Interpret Check Information

1. Select the check on the **Results List** or **Source** pane.
2. View the message on the **Result Details** pane.

   The message explains why the block of code is unreachable.

   ID 6: Unreachable code
   Unreachable code
   If-condition always evaluates to true at line 47 (column 8).
   Block ends at line 51 (column 4)

3. A code block is usually unreachable when the condition that determines entry into the block is not satisfied. See why the condition is not satisfied.

   a. On the **Source** pane, place your cursor on the variables involved in the condition to determine their values.
   b. Using these values, see why the condition is not satisfied.

Note Sometimes, a condition itself is redundant. For example, it is always true or coupled:

- Through an `||` operator to another condition that is always true.
- Through an && operator to another condition that is always false.

For example, in the following code, the condition `x%2==0` is redundant because the first condition `x>0` is always true.

```
assert(x>0);
if(x>0 || x%2 == 0)
```

If a condition is redundant, instead of a block of code, the condition itself is marked gray.

Step 2: Determine Root Cause of Check

Trace the data flow for each variable involved in the condition.

In the following example, trace the data flow for `arg`.

```
void foo(void) {
    int x=0;
    .
    .
    bar(x);
    .
    .
}

void bar(int arg) {
    if(arg==0) {
        /*Block 1*/
    }
    else {
        /*Block 2*/
    }
}
```

You might find that `bar` is called only from `foo`. Since the only argument of `bar` has value 0, the `else` branch of `if(arg==0)` is unreachable.

Possible fix: If you do not intend to call `bar` elsewhere and know that the values passed to `bar` will not change, you can remove the `if-else` statement in `bar` and retain only the content of `Block 1`.

To trace the data flow, select the check and note the information on the **Result Details** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Result Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.
- Otherwise, for each variable involved in the condition, find previous instances and trace back to the root cause of check. For more information on common root causes, see “Step 3: Look for Common Causes of Check” on page 22-70.

  Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

<table>
<tr><th>Variable</th><th>How to Find Previous Instances of Variable</th></tr>
<tr><td>Local Variable</td><td>Use one of the following methods:
• Search for the variable.
1 Right-click the variable. Select Search For All References.
All instances of the variable appear on the Search pane with the current instance highlighted.
2 On the Search pane, select the previous instances.
• Browse the source code.
1 Double-click the variable on the Source pane.
All instances of the variable are highlighted.
2 Scroll up to find the previous instances.</td></tr>
<tr><td>Global Variable
Right-click the variable. If the option Show In Variable Access View appears, the variable is a global variable.</td><td>1 Select the option Show In Variable Access View.
On the Variable Access pane, the current instance of the variable is shown.
2 On this pane, select the previous instances of the variable.
Write operations on the variable are indicated with ◀ and read operations with ▶.</td></tr>
<tr><td>Function return value
<code>ret=func();</code></td><td>1 Find the function definition.
Right-click <code>func</code> on the Source pane. Select Go To Definition, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration.
2 In the definition of <code>func</code>, identify each <code>return</code> statement. The variable that the function returns is your new variable to trace back.</td></tr>
</table>

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 22-77.

Step 3: Look for Common Causes of Check

Look for common causes of the **Unreachable code** check.

- Look for the following in your `if` tests:
  - You are testing the variables that you intend to test.

    For example, you might have a local variable that shadows a global variable. You might be testing the local variable when you intend to test the global one.
  - You are using parentheses to impose the sequence in which you want operations in the `if` test to execute.

  For example, `if((!a && b) || c)` imposes a different sequence of operations from `if(!(a && b) || c)`. Unless you use parentheses, the operations obey operator precedence rules. The rules can cause the operations to execute in a sequence that you did not intend.
- You are using = and == operators in the right places.

  Possible fix: Correct errors if any.

  - Use Polyspace Bug Finder to check for common defects such as `Invalid use of = operator` and `Variable shadowing`.
  - To avoid errors due to incorrect operation sequence, check for violations of `MISRA C:2012 Rule 12.1`.

- See if you are performing a test that you have performed previously.

  The redundant test typically occurs on the argument of a function. The same test is performed both in the calling and called function.

```
void foo(void) {
    if(x>0)
        bar(x);
    .
    .
}

void bar(int arg) {
    if(arg==0) {
    }
}
```

  Possible fix: If you intend to call `bar` later, for example, in yet unwritten code, or reuse `bar` in other programs, retain the test in `bar`. Otherwise, remove the test.
- See if your code is unreachable because it follows a `break` or `return` statement.

  Possible fix: See if you placed the `break` or `return` statement at an unintended place.
- See if the section of unreachable code exists because you are following a coding standard. If so, retain the section.

  For example, the `default` block of a `switch-case` statement is present to capture abnormal values of the `switch` variable. If such values do not occur, the block is unreachable. However, you might violate a coding standard if you remove the block.
- See if the unreachable code is related to an orange check earlier in the code. Following an orange check, Polyspace normally terminates execution paths that contain an error. Because of this termination, code following an orange check can appear gray.

  For example, Polyspace places an orange check on the dereference of a pointer `ptr` if you have not vetted `ptr` for `NULL`. However, following the dereference, it considers that `ptr` is not `NULL`. If a test `if(ptr==NULL)` follows the dereference of `ptr`, Polyspace marks the corresponding code block unreachable.

  For more examples, see:

  - "Gray Check Following Orange Check" on page 32-11

    An exception to this behavior is overflow. If you specify the appropriate **Overflow mode for signed integer** or **Overflow mode for unsigned integer**, Polyspace wraps the result of an

overflow and does not terminate the execution paths. See `Overflow mode for signed integer (-signed-integer-overflows)` or `Overflow mode for unsigned integer (-unsigned-integer-overflows)`.

- “Left operand of left shift may be negative”

Possible fix: Investigate the orange check. In the above example, investigate why the test `if(ptr==NULL)` occurs after the dereference and not before.

Review and Fix User Assertion Checks

This topic describes how to systematically review the results of a **User assertion** check in Polyspace Code Prover.

Follow one or more of these steps until you determine a fix for the **User assertion** check. There are multiple ways to fix this check. For a description of the check and code examples, see `User assertion`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2 or “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2.

How to use this check: Typically you use `assert` statements during debugging to check if a condition is satisfied at the current point in your code. For instance, if you expect a variable `var` to have values in a range `[1,10]` at a certain point in your code, you place the following statement at that point:

```
assert(var >=1 && var <= 10);
```

Polyspace statically determines whether the condition is satisfied.

Therefore, you can judiciously insert `assert` statements that test for requirements from your code.

- A red result for the **User assertion** check indicates that the requirement is definitely not met.
- An orange result for the **User assertion** check indicates that the requirement is possibly not met.

Step 1: Determine Root Cause of Check

Trace the data flow for each variable involved in the `assert` statement.

In the following example, trace the data flow for `myArray`.

```
int* getArray(int numberOfElements) {
    .
    .
    arrayPtr = (int*) malloc(numberOfElements);
    .
    .
    return arrayPtr;
}
void func() {
    int* myArray = getArray(10);
    assert(myArray!=NULL);
    .
    .
}
```

In this example, it is possible that in `getArray`, the return value of `malloc` is not checked for `NULL`.

Possible fix: If you expect a certain requirement, see if you have tests that enforce the requirement. In this example, if you expect `getArray` to return a non-`NULL` value, in `getArray`, test the return value of `malloc` for `NULL`.

To trace the data flow, select the check and note the information on the **Result Details** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Result Details** pane shows the line number of probable cause for the check, right-click in the **Source** pane. Select **Go To Line**. Enter the line number.
- Otherwise, for each variable involved in the condition, find previous instances and trace back to the root cause of the check. For more information on common root causes, see “Step 3: Look for Common Causes of Check” on page 22-70.

  Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 Double-click the variable on the **Source** pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with ◀ and read operations with ▶.

Variable	How to Find Previous Instances of Variable
Function return value `ret=func();`	**1** Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration. **2** In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 22-77.

Step 2: Look for Common Causes of Check

1. If the check is orange and occurs in a function, see if the function is called multiple times. Determine if the assertion fails only on certain calls. For those calls, navigate to the caller body and see if you have tests that enforce your assertion requirement.

   - To see the callers of a function, select the function name on the **Source** pane. All callers appear on the **Call Hierarchy** pane. Select a caller name to navigate to it in your source.
   - To determine if a subset of calls cause the orange check, use the option `Sensitivity context (-context-sensitivity)`. For a tutorial, see “Identify Function Call with Run-Time Error” on page 22-44.
2. If you have tests that enforce the assertion requirement, see if:

   - The assertion statement is within the scope of the tests.
   - You modify the test variables between the test and the assertion.

   For instance, the test `if(index < SIZE)` enforces that `index` is less than `SIZE`. However, the assertion `assert(index < SIZE)` can fail if:

   - It occurs outside the `if` block.
   - Before the assertion, you modify `index` in the `if` block.

   Possible fix: Consider carefully whether you must meet your assertion requirements. If you must meet those requirements, place tests that enforce your requirement. After the tests, avoid modifying the test variables.

Step 3: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2 or “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2.

For instance, after a function call, you assert a relation between two variables. Then:

1. Depending on the depth of the function call and the complexity of your code, Polyspace can sometimes approximate the variable ranges. Because of the approximation, the software cannot establish if the relation holds and produces an orange **User assertion** check.
2. Run dynamic tests on the orange check to determine if the assertion fails.
3. Try to reduce your code complexity and rerun the verification. Otherwise, add a comment and a justification in your result or code describing why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Note Before justifying an orange check, consider carefully whether you can improve your coding design.

Find Relations Between Variables in Code

This tutorial shows how to determine if the variables in an arbitrary operation in your code are previously related.

For instance, consider this operation:

```
return(var1 - var2);
```

- In your IDE, you can place breakpoints to stop execution and determine the values of `var1` and `var2` *for a specific run*.
- In Polyspace, after you analyze your code, the tooltips on `var1` and `var2` show their range of values *for all runs* that the verification considers.

However, the range information is not enough to determine if the variables are related. You must perform additional steps to determine the relation.

Insert Pragma to Determine Variable Relation

In this example, consider the operation highlighted. You cannot tell from a quick glance if `wheel_speed` and `wheel_speed_old` are related. However, this information is crucial to understand a possible bug in subsequent operations.

```
#define MAX_SPEED 120
#define TEST_TIME 10000

int wheel_speed;
int wheel_speed_old;

int out;

int update_speed(int new_speed) {
    if(new_speed < MAX_SPEED)
    return new_speed;
    else
    return MAX_SPEED;
}

void increase_speed(void)
{

  int temp, index=1;

  while(index<TEST_TIME) {
    temp = wheel_speed - wheel_speed_old;

    if(index > 1) {
       if (temp < 0)
         out = 1;
       else
         out = 0;
    }

    wheel_speed_old = update_speed(wheel_speed);
    index++;
  }
```

```
}
```

To understand why you need the relation between `wheel_speed` and `wheel_speed_old` and how to find the relation:

1. Constrain the range of the variable `wheel_speed` to an initial value of 0..100 for the Polyspace analysis. Use the analysis option `Constraint setup (-data-range-specifications)`.
2. Run analysis on this code and open the results. Select the gray **Unreachable code** check.

   ```
   if (temp < 0)
           out = 1;
   ```

   The check indicates that the variable `temp` is nonnegative. `temp` comes from the previous operation:

   ```
   temp = wheel_speed - wheel_speed_old;
   ```
3. See the range of `wheel_speed` and `wheel_speed_old`. Place your cursor on these variables. You see these ranges:

   - `wheel_speed`: 0..100
   - `wheel_speed_old`: Full range of an `int` variable.

   It is not clear from these ranges why `wheel_speed - wheel_speed_old` is always nonnegative. You have to find out if the variables are somehow related.
4. Insert a pragma before the line where you want the variable relation. Add the following line just before `if(temp < 0)`:

   ```
   #pragma Inspection_Point wheel_speed wheel_speed_old
   ```
5. Rerun the analysis and open the results. Place your cursor on `wheel_speed_old` in the line that you added.

   The tooltip confirms that `wheel_speed` and `wheel_speed_old` are related:

   ```
   wheel_speed_old <= wheel_speed
   ```
6. To find how the two variables got related, search for all instances of `wheel_speed_old`. On the **Source** pane, right-click `wheel_speed_old` and select **Search For All References**.

   Browse through the instances. In this case, you see that the line which relates `wheel_speed` and `wheel_speed_old` is:

   ```
     wheel_speed_old = update_speed(wheel_speed);
   ```

   This line ensures that after the first run of the `while` loop, `wheel_speed_old` is less than or equal to `wheel_speed_old`. The branch `if(index > 1)` is entered from the second run onwards. In this branch, the relation between `wheel_speed` and `wheel_speed_old` is reflected through the gray **Unreachable code** check.

Tip Ignore the details of the relation shown in the tooltip. Use the tooltip to confirm if certain variables are related. Then, search for instances of the variable to find how they are related.

Further Exploration

You can use the pragma `Inspection_Point` to determine the relation between variables at any point in the code. You can enter as many variables as you want in the `#pragma` statement:

```
#pragma Inspection_Point var1 var2 ... varn
```

Try this technique on other examples. For instance, select **Help** > **Examples** > **Code_Prover_Example.psprj**. Group the results by file. In the file `single_file_analysis.c`, you see this code:

```
    if (output_v7 >= 0) {

      #pragma Inspection_Point output_v7 s8_ret
      saved_values[output_v7] = s8_ret;
      return s8_ret;
    }
```

If you place your cursor on `s8_ret` in the last two statements, you find that the ranges of `s8_ret` are different. Find out what changed between the two statements.

Hint: The tooltip in the `#pragma` statement indicates that the variable `s8_ret` is related to the variable `output_v7`. Note the orange check that reduces the range of `output_v7`.

See Also

Related Examples

- “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2
- “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2

Review Polyspace Results on AUTOSAR Code

This topic describes a component-based approach to verifying AUTOSAR code with Polyspace. For an integration analysis approach, see "Choose Between Component-Based and Integration Analysis of AUTOSAR Code with Polyspace" on page 8-5.

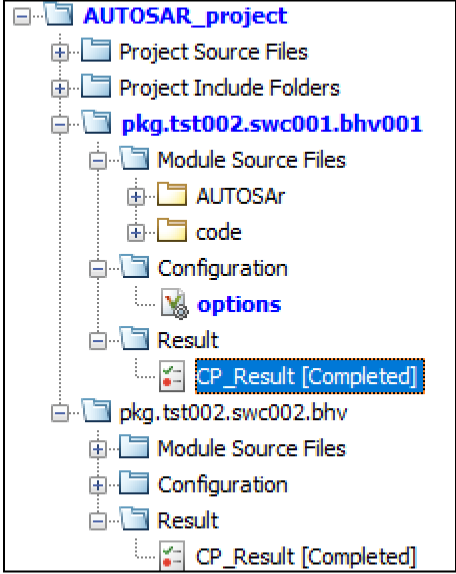
This tutorial describes how to open Polyspace Code Prover results for AUTOSAR-specific code and interpret results that highlight violation of data constraints in the ARXML.

Code Prover checks the code implementation of AUTOSAR software component-s for mismatch with specifications in the ARXML. For instance, if an RTE function argument has a value outside the constrained range defined in the ARXML, the analysis flags a possible issue.

> To follow the steps in this tutorial, run Polyspace on the demo files in `polyspaceroot\help\toolbox\codeprover\examples\polyspace_autosar`. Use the information in this tutorial to review the AUTOSAR-specific results. For help on running analysis, see "Run Polyspace on AUTOSAR Code" on page 8-14.

Open Results

If you run the analysis in the Polyspace user interface, you can open each result directly. Double-click the result that you want to open.



If you run the analysis by using scripts, after analysis, you can open the results in several ways.

- Open the file `psar_project.xhtml` from your project folder in a web browser. You see an overview of results for all software components and navigate to results for each software

component. For more information, see "See Overview of Results for all Software Components" on page 22-81.

- Open the file psar_project.psprj from your project folder in the Polyspace user interface. Open each result on the **Project Browser** pane.
- Navigate to the folders containing the individual results files. Open a result file (with extension .pscp) in the Polyspace user interface.

  The results files are stored in a subfolder AUTOSAR of the project folder. The path to each result follows the fully qualified name of the internal behavior of the software component. For instance, for a fully qualified name pkg.component.bhv, the results are stored in AUTOSAR\pkg\component\bhv\verification (the final subfolder is named CP_Result if you run a verification in the Polyspace user interface).

See Overview of Results for all Software Components

Before opening a specific result set, you might want to see an overview of results for all software components. Do one of the following:

- Open the file psar_project.xhtml in the project folder on the machine where you run the analysis. If you are reviewing results from a different machine, you might not have access to this file.
- Upload the result files to Polyspace Access. To begin, see "Upload Results to Polyspace Access" on page 2-28 and "Review Polyspace Code Prover Results in Web Browser".

Use the first method for easier understanding of results.

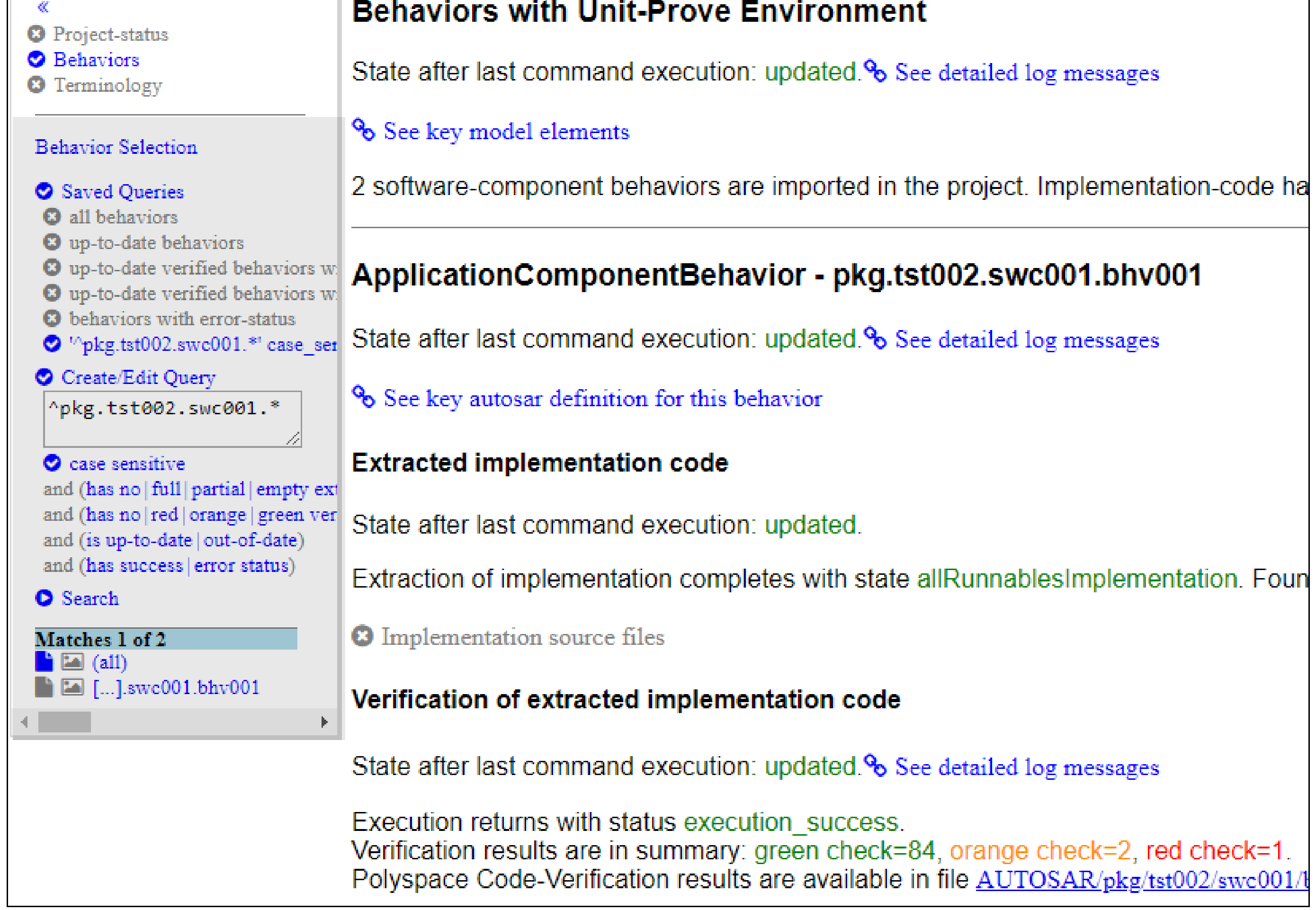
In the file psar_project.xhtml, click the ☰ icon on the upper left. On the left pane, click **Behaviors**. You can see the list of all software components whose internal behavior-s are extracted.

You can filter this list to display only the software components that you are interested in. To see specific software components, in the search box, enter the fully qualified name of the software component that you are interested in.

You can also enter regular expressions to see multiple components. For instance, to see all components whose qualified names begin with pkg.tst002.swc001, enter the expression:

```
^pkg.tst002.swc001.*
```

Click **Search**. The list on the right displays only the software components that you queried for.



You can also filter out components based on other criteria:

- Success or failure of verification

  To see only software components that completed verification, click and then clear the **error status** filter.
- Presence or absence of certain kinds of results, for instance, red checks

  To see only software components that have red checks, click everything on the row containing the **red** filter except the **red** filter itself.

See Runnables and Source Files in Software Component

For each software component, you can see this information in the file `psar_project.xhtml` in your project folder (see the preceding figure).

- The state of this software component with respect to the analysis. That is, whether the software component specification was parsed, its source code extracted, and then analyzed with Code Prover.

  To make sure that the Code Prover analysis was complete, under the section **Verification of extracted implementation code**, look for this statement:

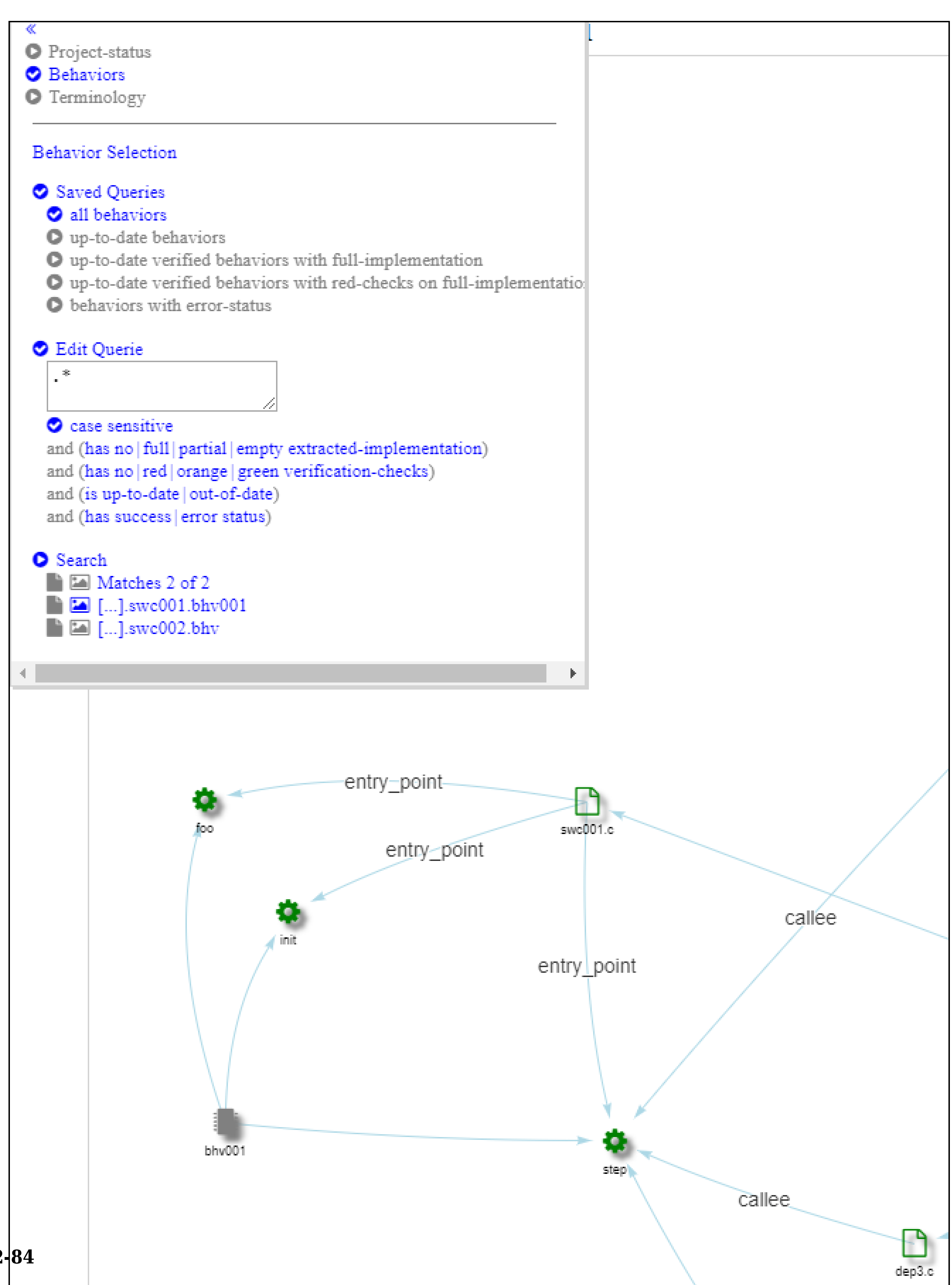
  ```
  State after last command execution: updated.
  ```
- Functions provided by this software component and the `Rte_` functions used.

  To see this list, click the link:

  ```
  See key autosar definition for this behavior
  ```
- Graphical view of runnables in the software component. The graphical view shows:

  - Entry-point functions implementing the runnables and their callees
  - Files containing these functions

  To see this view, in the list of software components on the left pane, click the (behavior graph) icon for the software component you are interested in. To return from the graph to the textual description of the software component, click the (behavior page) icon.



«
Project-status
Behaviors
Terminology
Behavior Selection
Saved Queries
all behaviors
up-to-date behaviors
up-to-date verified behaviors with full-implementation
up-to-date verified behaviors with red-checks on full-implementatio
behaviors with error-status
Edit Querie
.*
case sensitive
and (has no | full | partial | empty extracted-implementation)
and (has no | red | orange | green verification-checks)
and (is up-to-date | out-of-date)
and (has success | error status)
Search
Matches 2 of 2
[...].swc001.bhv001
[...].swc002.bhv
entry_point
foo
swc001.c
entry_point
init
callee
entry_point
bhv001
step
callee
dep3.c

In this example, you see that the software component with internal behavior `bhv001` has three runnables implemented through the entry-point functions `foo`, `init`, and `step`. All three entry-point functions are defined in the file `swc001.c`.

The function `step` calls functions defined in other files, for instance, `dep3.c`. You can click the icon for `step` to see only the files involved in the implementation of the runnable `step`. To revert to the full graphical view of the software component, click anywhere in the blank space in the graph.

- Overview of Code Prover results with links to the result files.

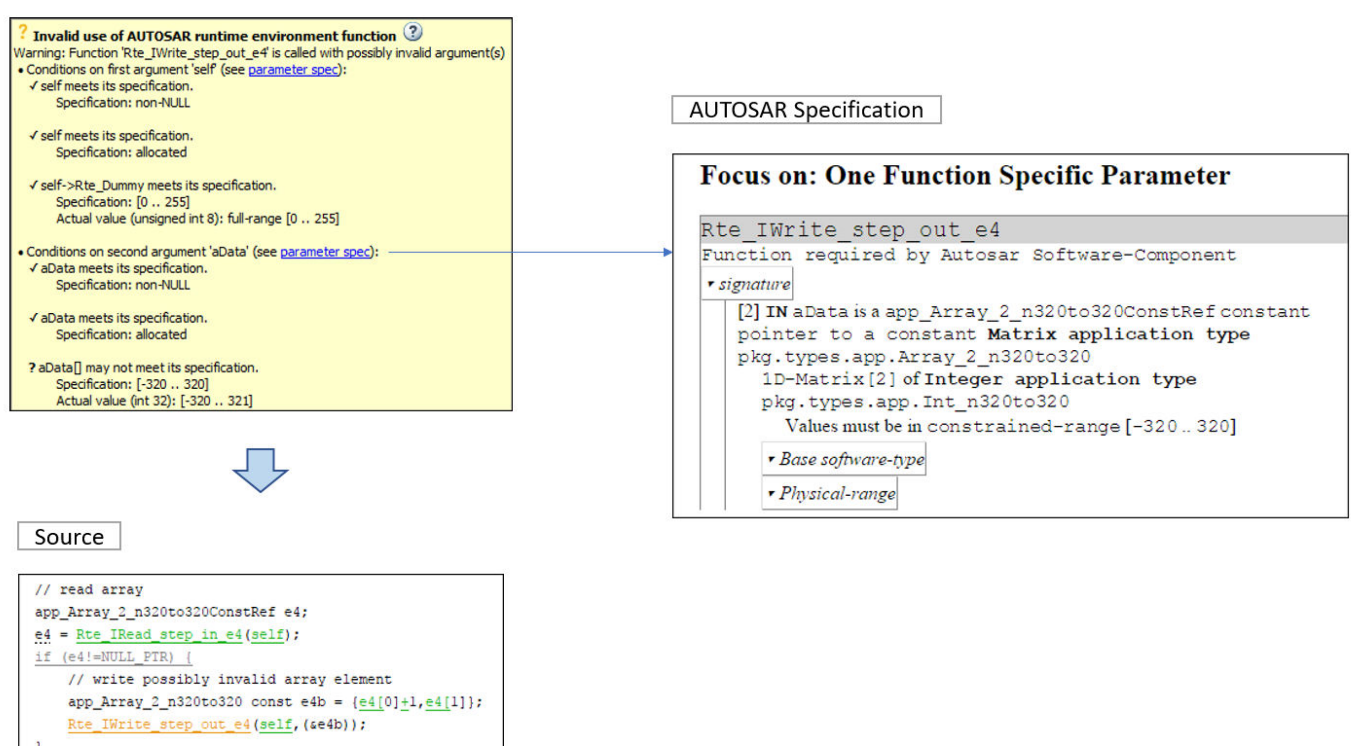
  Look for lines like these lines:

  ```
  Verification results are in summary: green check=84, orange check=2, red check=1
  ```

  Click the link following the line to open the result file in the Polyspace user interface. If you haven't opened a `.pscp` file before, clicking the link might simply download the result file. Make sure that `.pscp` files always open in the Polyspace user interface (with the executable *`polyspaceroot`*`\polyspace\bin`, where *`polyspaceroot`* is the Polyspace installation folder).

  The results consist of AUTOSAR-specific run-time checks such as `Invalid result of AUTOSAR runnable implementation` and general C/C++ run-time checks such as `Division by zero`.

Interpret AUTOSAR Specific Run-time Checks for Software Component



On the **Results List** pane, select the result **Invalid result of AUTOSAR runnable implementation** or **Invalid use of AUTOSAR runtime environment function**. Investigate the result further by using the information on various panes.

Check Return Value and Arguments

Using the information on the **Result Details** pane, determine whether the return value or an argument violates data constraints in the ARXML or can be NULL-valued. Look for the **!** icon that indicates a definite error or the **?** icon that indicates a possible error.

For the return value and each argument, you see the actual possible values at run time and the values allowed by the data type in the ARXML specification. Compare them and find the value that is not allowed.

The result **Invalid result of AUTOSAR runnable implementation** determines if the return value of the function implementing the runnable or the output arguments can violate the data constraints. The result **Invalid use of AUTOSAR runtime environment function** determines if the input arguments to an `Rte_` function violates data constraints.

Check Argument Spec (Optional)

Sometimes, you might want to see the Application Data Type from which the variable Base Software Type originates. Click the blue parameter spec link and see the ARXML extract that describes this information about the parameter or return value data type:

- Application Data Type, Implementation Data Type, and Base Software Type
- Data Constraint, Unit, and Computation Method

Find Root Cause of Result

Investigate how the variable acquires the values that violate the data constraints. To trace back in your code, on the **Source** pane, right-click a variable and search for all its instances or navigate to its definition. For more tips, see “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2 or “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2.

Decide whether to fix your code or ARXML, or justify the result through comments. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2 or “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2.

See Also

`Invalid result of AUTOSAR runnable implementation` | `Invalid use of AUTOSAR runtime environment function`

More About

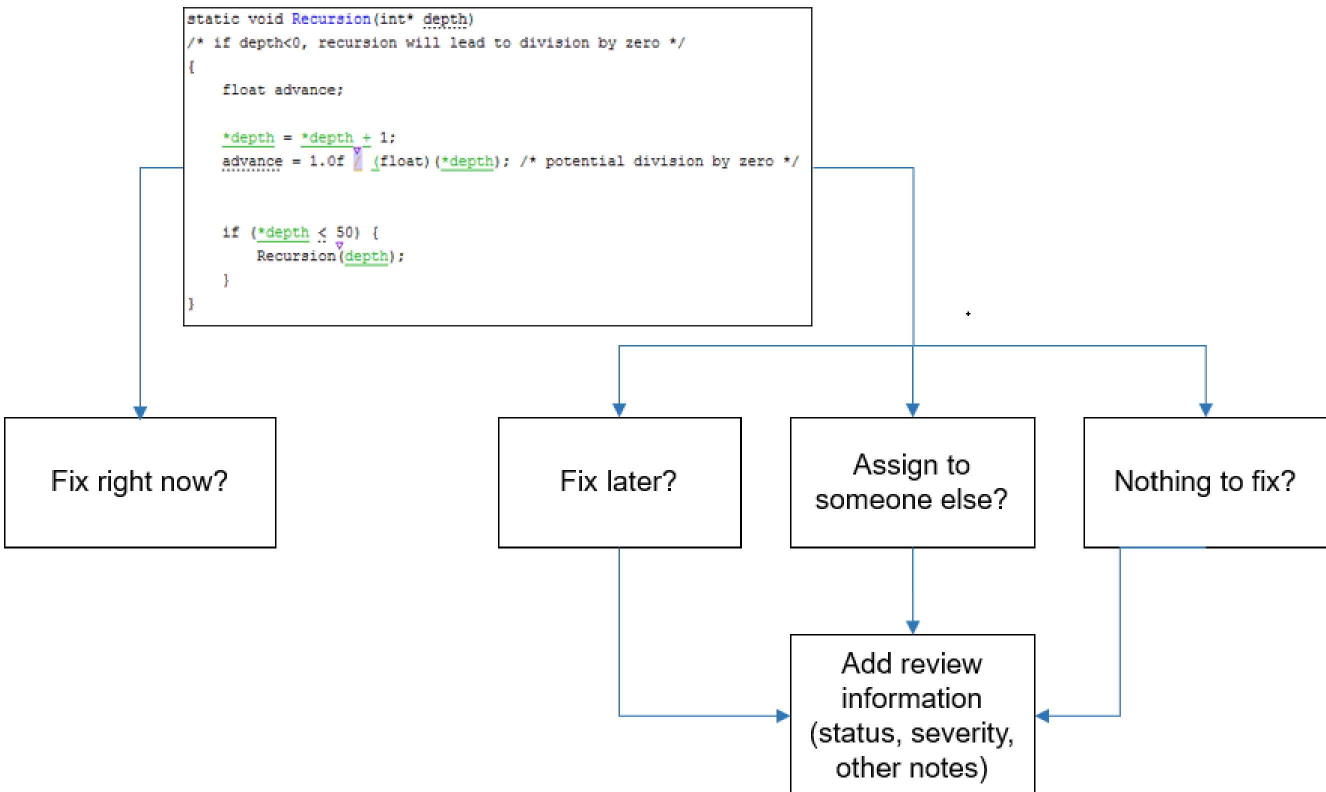
- “Benefits of Polyspace for AUTOSAR” on page 8-7
- “Run Polyspace on AUTOSAR Code” on page 8-14

Fix or Comment Polyspace Results

Address Results in Polyspace User Interface Through Bug Fixes or Justifications

This topic describes how to add review information to Polyspace results in the user interface of the Polyspace desktop products. For a similar workflow in the Polyspace Access web interface, see *“Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2.*

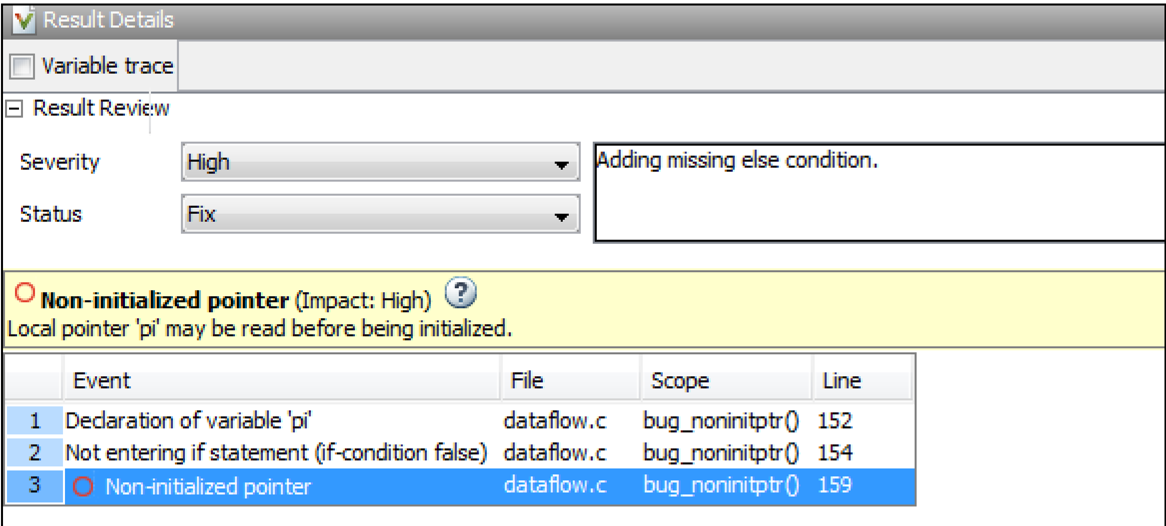
Once you understand the root cause of a Polyspace finding, you can fix your code. Otherwise, add review information to your Polyspace results to fix the code later or to justify the result. You can use the information to keep track of your review progress.



If you add review information to your results file, they carry over to the results of the next analysis on the same project. If you add the same information as comments to your code (annotate), they carry over to any subsequent analysis of the code, whether in the same project or not. You can also hide results using code annotations.

Add Review Information to Results File

You can add review information either on the **Results List** or **Result Details** pane. Select a result, then set the **Severity** and **Status** fields, and optionally, enter notes with more explanations.



The status indicates your response to the Polyspace result. If you do not plan to fix your code in response to a result, assign one of the following statuses:

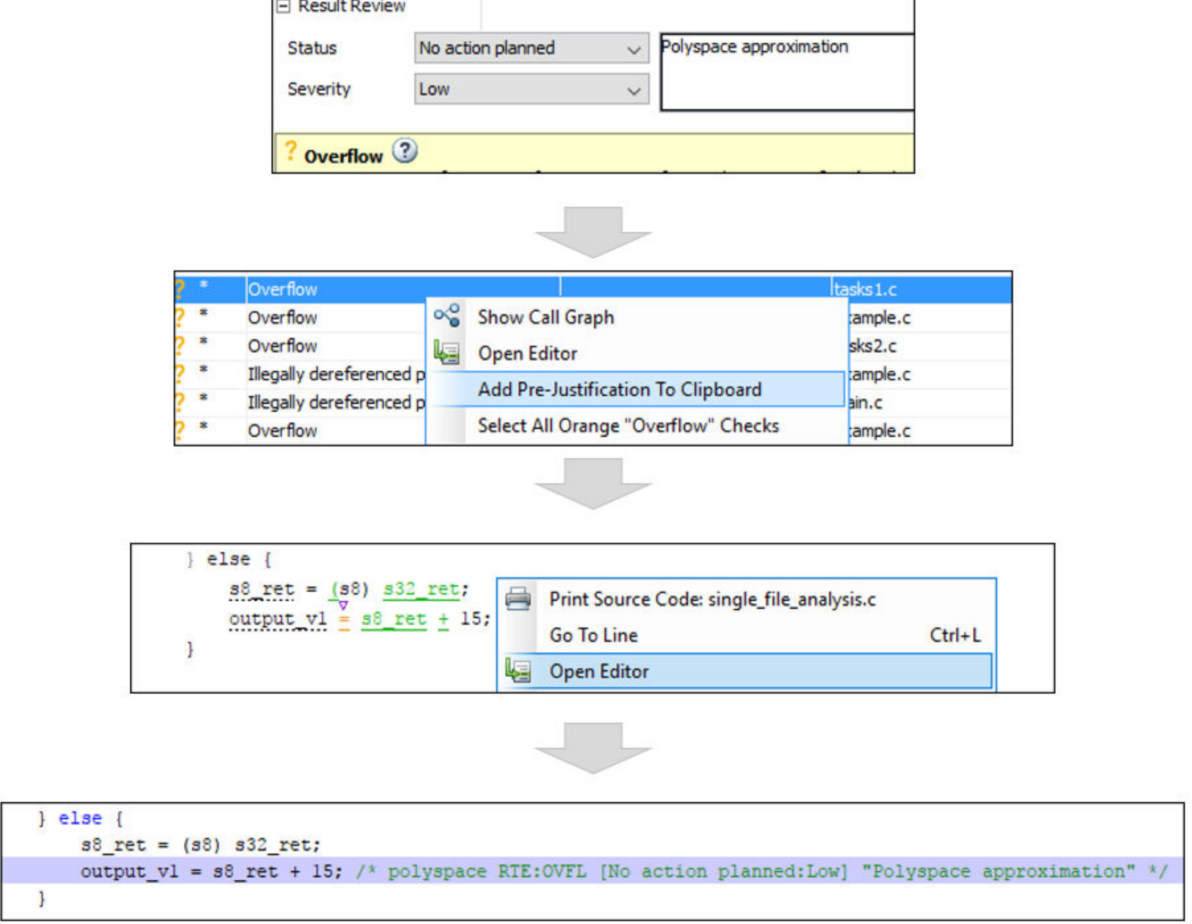
- `Justified`
- `No Action Planned`
- `Not a Defect`

These statuses indicate that you have given due consideration and justified that result (retained the code despite the result). Note that subsequent analyses continue to show justified results as before. For instance, a Code Prover result that was previously orange does not turn green after justification. However, during review, you can filter out justified results in one click and focus only on results that are not justified. See “Filter and Group Results in Polyspace Desktop User Interface” on page 24-2.

You can also create your own statuses to assign. See “Create Custom Review Status” on page 2-27.

Comment or Annotate in Code

You can also add specific code comments or annotations in response to Polyspace results. If you enter code comments or annotations in a specific syntax, on the next analysis of the code, the software can read them and populate the **Severity**, **Status**, and **Comment** fields in the result details.



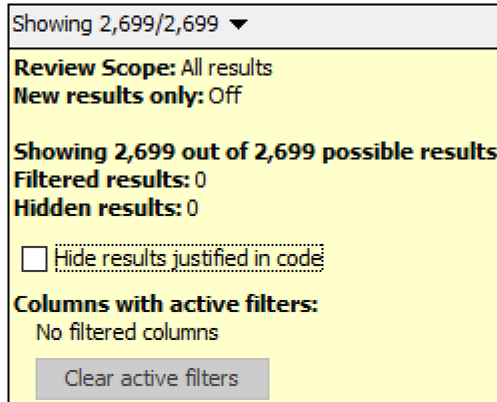
You can either type the annotation directly or copy it from the user interface:

- In the user interface, to copy annotations, right-click a result and select **Add Pre-Justification To Clipboard**. Open your source code in an editor and paste *on the same line as* the result.
- Type the annotation on *on the same line as* the result. See the annotation syntax in “Annotate Code and Hide Known or Acceptable Results” on page 30-2.

If you copy or type the annotation without explicitly assigning a status, Polyspace assumes that you have set a status of `No Action Planned`. The software hides the result from all places (except reports needed for certification[4]). The only exceptions are the safety-critical Code Prover run-time checks, which are hidden from the results list but not the source code. If you want to explicitly set a status, first fill the **Status** field for a result and then copy the annotation to your code. Paste on the line containing the result.

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

4 Reports generated from Polyspace results are typically meant for archiving and certification. Therefore, the reports contain all Polyspace results, justified or otherwise. Justified results show the justification status, for instance, `No Action Planned`, along with comments supporting the justification. These reports allow standards committees such as certification authorities to verify if a Polyspace result was justified for approved reasons.



See Also

More About

- "Annotate Code and Hide Known or Acceptable Results" on page 30-2
- "Import Review Information from Previous Polyspace Analysis" on page 20-2

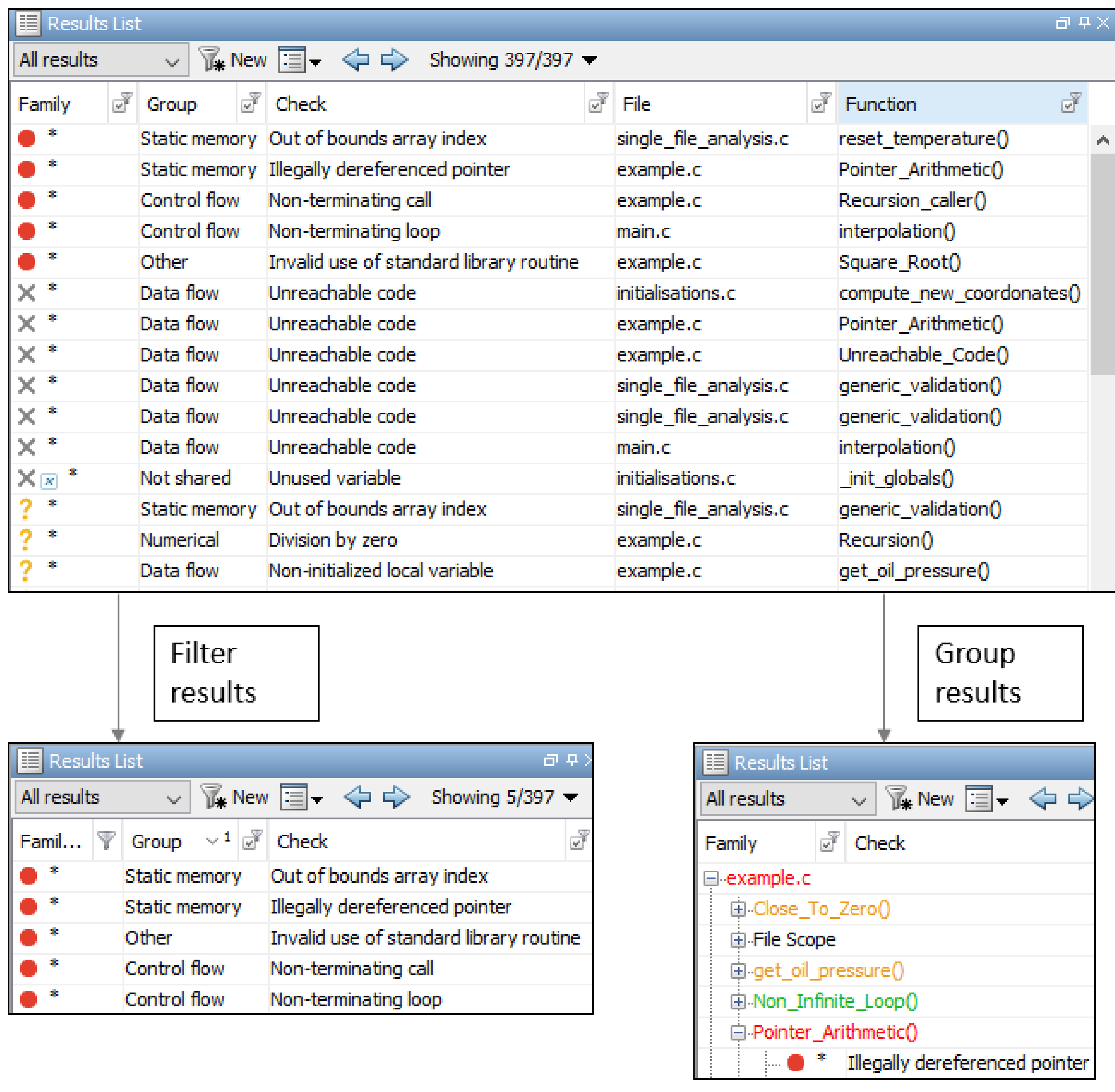
Manage Results

- “Filter and Group Results in Polyspace Desktop User Interface” on page 24-2
- “Prioritize Check Review” on page 24-9

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" on page 28-8.

When you open the results of a Polyspace analysis, you see a flat list of defects (Bug Finder), run-time checks (Code Prover), coding rule violations or other results. To organize your review, you can narrow down the list or group results by file or result type.



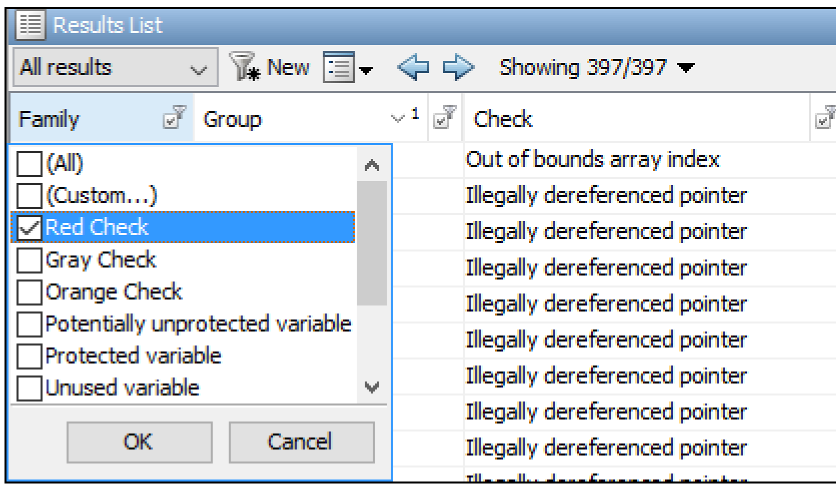
Some of the ways you can use filtering are:

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

Filter Results

Filter Using Results List



You can filter using the columns on the **Results List** pane. Click the icon on the column headers to see the available filters. For instance:

- To see only Bug Finder defects with high impact, from the filters on the **Information** column, clear all check boxes except **Impact: High**.
- To see only results that are not yet justified, clear the **True** filter on the **Justification** column. This column might not be visible by default. To see the column, right-click any column header and select **Justification**.

  For information on justification, see “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2.

For information on the columns, see:

- “Results List in Polyspace Desktop User Interface”
- “Results List in Polyspace Desktop User Interface” on page 21-12

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”.

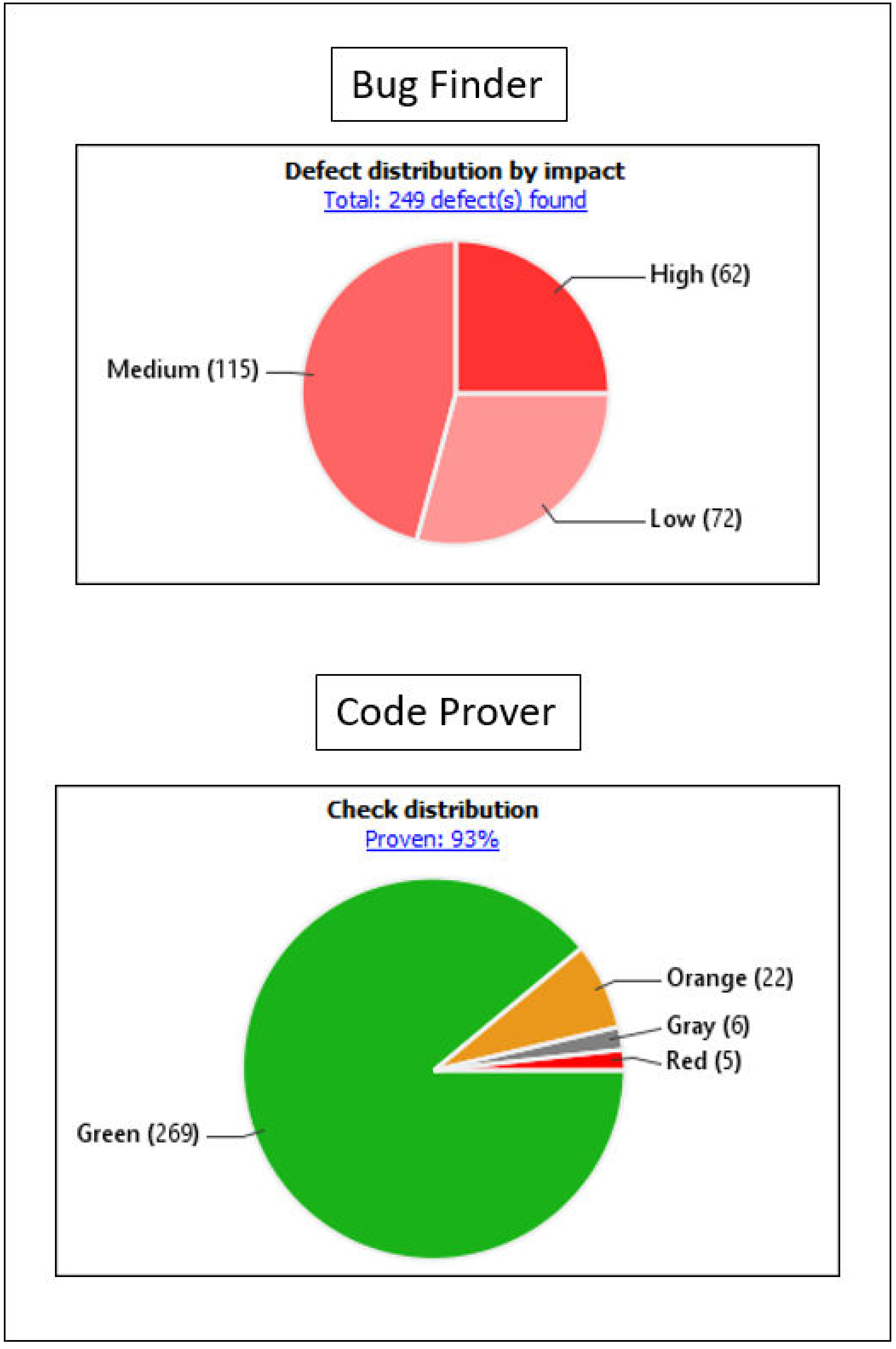
If you do not want to filter by the exact contents of a column, you can use a custom filter instead. For instance, you want to filter out subfolders of a specific folder. Instead of filtering out each subfolder in

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

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

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

Filter Using Dashboard



You can click graphs on the **Dashboard** pane to filter results. For instance:

- To see only high-impact defects in Bug Finder, click the corresponding section of the **Defect distribution by impact** chart.
- To see only red checks in Code Prover, click the corresponding section of the **Check distribution** chart.

To see all results again, click the link **View all results in this scope**.

Filter Using Orange Sources

An orange source can cause multiple orange checks in Code Prover. You can display all orange checks from the same source and review them together.

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

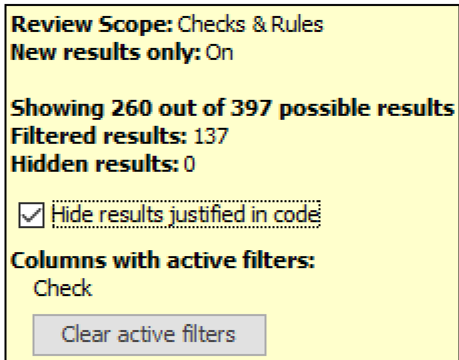
```
void func (int input) {
int val1;
double val2;
val1 = input++;
val2 = 1.0/input;
}
```

To begin, select **Window** > **Show/Hide View** > **Orange Sources**. You see the list of orange sources. Select an orange source to see all orange checks coming from this source.

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_i	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).

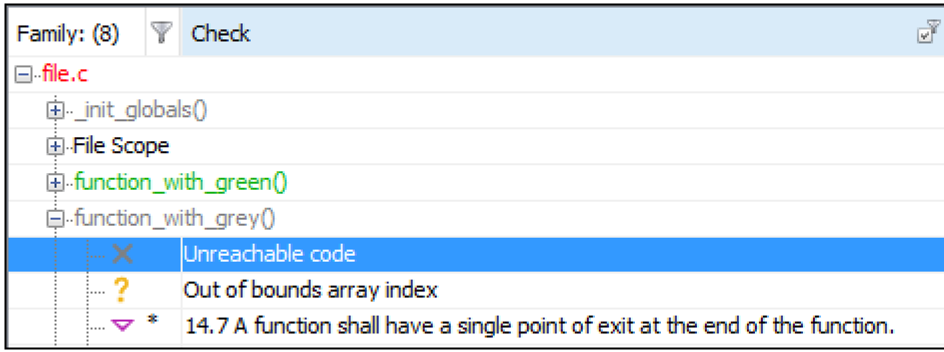
You see this information about the filters:

- **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 Results in Polyspace User Interface Through Bug Fixes or Justifications" on page 23-2.
- **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".
  - The global variables (Code Prover) are grouped by their usage. For more information, see "Global Variables".
  - 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

- “Prioritize Check Review” on page 24-9

Prioritize Check Review

This example shows how to prioritize your check review. Try the following approach. You can also develop your own procedure for organizing your orange check review.

Tip For easier review, run Polyspace Bug Finder on your source code first. Once you address the defects that Polyspace Bug Finder finds, run Polyspace Code Prover on your code.

---

1. Before beginning your check review, do the following:

   - See the **Code covered by verification** graph on the **Dashboard** pane. See if the **Files**, **Functions** and **Code operations** columns display a value closer to 100%. Otherwise, identify why Polyspace could not cover the code.

     For more information, see "Reasons for Unchecked Code" on page 34-77. If a substantial number of functions or code operations were not covered, after identifying and fixing the cause, run verification again.
   - See if you have used the right configuration. Select the link **View configuration for results** on the **Dashboard** pane.

     Sometimes, especially if you are switching between multiple configurations, you can accidentally use the wrong configuration for the verification.
2. From the drop-down list in the left of the **Results List** pane toolbar, select **Critical checks**.

   This action retains only red, gray and critical orange checks.
3. Click the forward arrow to go to the first unreviewed check. Review this check.

   For more information, see "Interpret Code Prover Results in Polyspace Desktop User Interface" on page 21-2.

   Continue to click the forward arrow until you have reviewed through all of the checks.
4. Before reviewing orange checks, review red and gray checks.
5. Prioritize your orange check review by:

   - Files and functions: For easier review, begin your orange check review from files and functions with *fewer* orange checks.

     To view the percentage of non-orange checks per file and function, on the **Results List** pane, from the list, select **File**. Right-click a column header and select **%**.
   - Check type: Review orange checks in the following order. Checks are more difficult to review as you go down this order.

Review Order	Checks
First	• `Out of bounds array index` • `Non-initialized local variable` • `Division by zero` • `Invalid shift operations`
Second	• `Overflow` • `Illegally dereferenced pointer`
Third	Remaining checks

- Orange check sources: Review all orange checks caused by a single variable or function. Orange checks often arise from variables whose values cannot be determined from the code or functions that are not defined.

  To review the top sources, view the **Top 5 orange sources** graph on the **Dashboard** tab or the **Orange Sources** tab. You can also select an orange source on either tab to see only the orange checks caused by the source. For more information, see “Filter Using Orange Sources”.
- Result details: Review all results that originate from the same cause. Sometimes, the **Detail** column on the **Results List** pane shows additional information about a result. For instance, if multiple issues trigger the same coding rule violation, this column shows the issue. Click the column header so that results that originate from the same type of issue are grouped together. Review the results in one go.

6. To ensure that you have addressed all red and critical orange checks, run verification again and view your results.
7. If you do not have red or unjustified critical orange checks, from the drop-down list in the left of the **Results List** pane toolbar, select **All results**.

   Depending on the quality level you want, you can choose whether to review the noncritical orange checks or not. For more information, see “Managing Orange Checks in Polyspace Code Prover” on page 33-5.
8. To see what percentage of checks you have justified:

   a. If you want the percentage broken down by color and type, on the **Results List** pane, from the list, select **Family**. If you want the percentage broken down by file and function, select **File**.
   b. View the entries in the **Justified** column.

See Also

Related Examples

- “Filter and Group Results in Polyspace Desktop User Interface”

Generate Reports from Polyspace Results

- “Generate Reports from Polyspace Results” on page 25-2
- “Export Polyspace Analysis Results” on page 25-5
- “Export Polyspace Analysis Results to Excel by Using MATLAB Scripts” on page 25-9
- “Export Global Variable List” on page 25-11
- “Visualize Code Prover Analysis Results in MATLAB” on page 25-14
- “Customize Existing Code Prover Report Template” on page 25-17
- “Sample Report Template Customizations” on page 25-22
- “Generate Report Containing MISRA C:2012 Violations, Code Metrics, and Runtime Check Results” on page 25-25

Generate Reports from Polyspace Results

This topic shows how to generate reports from results generated with a Polyspace desktop product. To generate reports from results uploaded to the Polyspace Access web server, see `polyspace-report-generator`.

To generate reports from Polyspace results, you can do one of the following:

- Run a Polyspace analysis and create a report from the analysis results. See the workflow described here.
- Specify that a report will be automatically generated after analysis. For more information on the options, see “Reporting”. Report generation immediately after analysis is supported for both desktop and server products.
- Export your results to a text file and generate graphs and statistics. See “Export Polyspace Analysis Results” on page 25-5.

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.[5]

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.

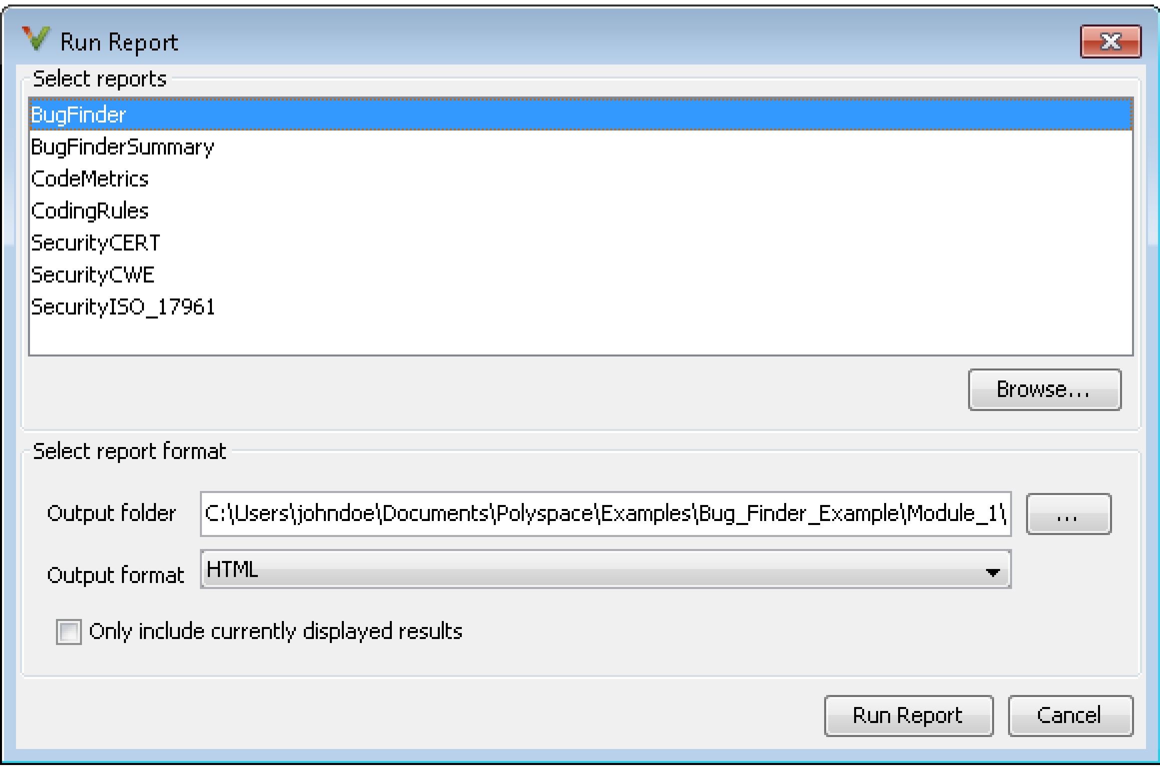
Generate Reports from User Interface

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.

5 Reports generated from Polyspace results are typically meant for archiving and certification. Therefore, the reports contain all Polyspace results, justified or otherwise. Justified results show the justification status, for instance, `No Action Planned`, along with comments supporting the justification. These reports allow standards committees such as certification authorities to verify if a Polyspace result was justified for approved reasons.



3 Select the following options:

- In the **Select Reports** section, select the types of reports that you want to generate. Press the **Ctrl** key to select multiple types. For example, you can select **BugFinder** and **CodeMetrics**.
- Select the **Output folder** in which to save the report.
- Select an **Output format** for the report.
- If the display language (Windows) or locale (Linux) of your operating system is set to another language, you see an option to generate English reports. Select this option if you want an English report, otherwise the report is in another language.
- If you want to filter results from your report, use filters on the **Results List** pane to display only the results that you want to report. Then, when generating reports, select **Only include currently displayed results**. You cannot display filtered reports for results downloaded from Polyspace Metrics.

  For more information on filtering, see “Filter and Group Results in Polyspace Desktop User Interface” on page 24-2.

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** HTMLreports 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 Code Prover Report Template" on page 25-17
- "Export Polyspace Analysis Results" on page 25-5

Export Polyspace Analysis Results

You can export your analysis results to a tab separated values (TSV) text file, a MATLAB table, or to a standard JSON format. Using the exported content, you can:

- Generate graphs or statistics about your results that you cannot readily obtain from the user interface by using MATLAB or Microsoft Excel. For instance, for each Code Prover check type (**Division by zero**, **Overflow**), you can calculate how many checks are red, orange, or green.
- Integrate the analysis results with other checks you perform on your code.

Export Results to Text File

You can export results to a tab delimited text file (TSV) from the user interface or command line.

The exported text file uses the character encoding on your operating system. If special characters from your comments are not exported correctly in the text file, change the character encoding on your operating system before exporting.

Export Results from User Interface (Desktop Products Only)

1. Open your analysis results.
2. Export all results or only a subset of the results.

   - To export all results, select **Reporting > Export > Export All Results**.
   - If you want to filter results from your report, use filters on the **Results List** pane to display only the results that you want to report. Then, when exporting results, select **Reporting > Export > Export Currently Displayed Results**.

     For more information on filtering, see “Filter and Group Results in Polyspace Desktop User Interface” on page 24-2.
3. Select a location to save the text file and click **OK**.

Note If you apply a review scope that sets thresholds for code metrics and you export all results, the generated file lists the results for the code metrics as **Green** (pass) or **Red** (fail) in the color column (third column). If you do not set thresholds for code metrics, the exported file shows **Not Applicable** for code metrics results in the color column.

Export Results From Command Line

Use the option `-format csv` with the `polyspace-results-export` command. For example, to generate a csv file from results file `C:\Polyspace_Workspace\myProject\Module_1\BF_Result\ps_results.psbf`, run this command:

```
polyspace-results-export -format csv -results-dir C:\Polyspace_Workspace\myProject\Module_1\BF_Result
```

Export Results to MATLAB Table

If you write MATLAB scripts to run Polyspace, you can read your Polyspace analysis results into a MATLAB table for further processing. See:

- “Visualize Bug Finder Analysis Results in MATLAB”
- “Visualize Code Prover Analysis Results in MATLAB” on page 25-14

Export Results to JSON Format

You can export Polyspace results to a JSON object. The JSON format follows the standard notation provided by the OASIS Static Analysis Results Interchange Format (SARIF).

Use the option `-format json-sarif` with the `polyspace-results-export` command. For more information, see `polyspace-results-export`.

The JSON format contains some additional information such as the checker short name and the full message that accompanies a result. Use the JSON format if you want to use this short name or message. You can also use this format for a more standardized reporting of results. For instance, if you use several static analysis tools and want to report their results in one interface by using a single parsing algorithm, you can export all the results to the standard SARIF JSON format.

View Exported Results

The exported results include the information available on the **Results List** pane in the desktop user interface or Polyspace Access web interface (except for line and column information). See:

- “Results List in Polyspace Desktop User Interface”
- “Results List in Polyspace Desktop User Interface” on page 21-12
- “Results List in Polyspace Access Web Interface” on page 26-17

Note that some **Results List** column headers might be labeled differently in the exported results file.

Some other differences in presentation between the **Results List** pane and exported results are listed below.

- The TSV file and MATLAB table contain these additional columns compared to the **Results List**:
  - **New** column — Shows whether a result is new compared to the previous run.
  - **Key** column — The entry in this column is based on the result family, result acronym, and the location of the result in the file. See also “Compare Merged Results Using Exported Keys” on page 25-6.
  - **URL** column (Polyspace Access results only) — Click the URL to open the corresponding result in the Polyspace Access interface.
- The JSON file stores the results as objects contained in a `results` array. Each object has a list of key-value pairs that store the results information, including whether the result is new compared to the last run ("`baselineState`") and the result key. For more on the JSON format see SARIF-v2.1.0.

You cannot identify the location of a Bug Finder result in your source code via the text file. However, you can still parse the file and generate graphs or statistics about your results.

Compare Merged Results Using Exported Keys

When you merge exported analysis results from multiple modules that contain common files, you can use the **Key** and other fields from the exported results to eliminate duplicates. For instance, if you

run coding-rule checking on two different modules and merge the results, coding rule violations in common header files appear twice in the results.

To eliminate duplicates, compare the **Key** and **File** values of the results. If two results have the same **Key** and **File** values, one is a duplicate of the other.

By default, each result key is based on the result family (for instance `numerical`), result short name (for instance `float_ovfl`), and the location of the result in the file. To generate more localized keys for results that are located inside a function body, use the `-key-mode function-scope` option with the `polyspace-results-export` or the `polyspace-report-generator` commands. The commands generate keys that are based on the result family, result short name, and the location of the result within the function body. You can then identify duplicates more accurately by also comparing the **Function** values.

In rare instances, results that have the same key and the same location inside a file or function body might not be duplicates. In those instances you need to compare the results manually, for instance by comparing the result messages in the user interface, to determine whether the results are duplicates or not.

Enable Function Scope for Exported Keys

To enable the function scope for exported keys:

- **At The Command Line**

  Enter either of these commands to export locally stored results:

  ```
  polyspace-results-export -results-dir folderPath -key-mode function-scope -format csv
  ```

  or

  ```
  polyspace-report-generator -generate-results-list-file -results-dir folderPath -key-mode function-scope
  ```

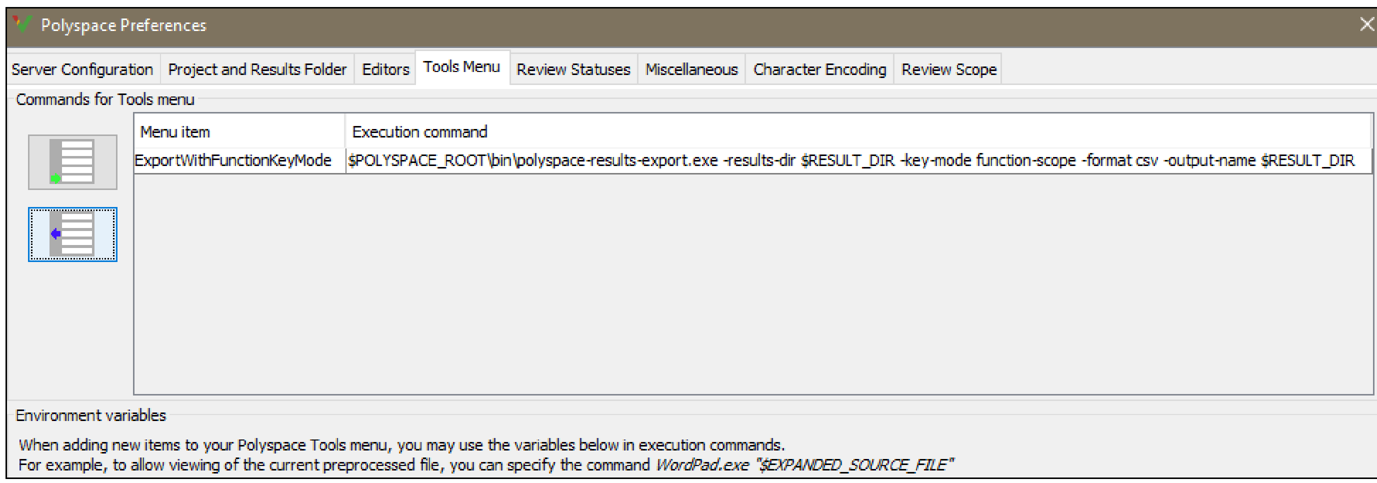
  where *`folderPath`* is the path of the folder that contains the Polyspace analysis results.

  Make sure that you run these commands from a location where you have write permissions, or use option `-output-name` to specify a location to store the generated file of exported results.

- **In the Polyspace Desktop User Interface**

  Create a menu item by going to **Tools** > **Preferences** and entering this command on the **Tools Menu** tab:

  ```
  $POLYSPACE_ROOT\bin\polyspace-results-export.exe -results-dir $RESULT_DIR
   -key-mode function-scope -format csv -output-name $RESULT_DIR
  ```



You can then export results by using the menu item you created from **Tools** > **External Tools**.

To export results with the default key mode (without function location), use the **Reporting** > **Export** menu.

When you export results with the function scope enabled, the key entries for results that are inside a function have a `FN` prefix.

See Also

`polyspace-results-export`

Related Examples

- “Visualize Code Prover Analysis Results in MATLAB” on page 25-14
- “Export Global Variable List” on page 25-11
- “Export Polyspace Analysis Results to Excel by Using MATLAB Scripts” on page 25-9

Export Polyspace Analysis Results to Excel by Using MATLAB Scripts

You can export the results of a Bug Finder or Code Prover analysis to an Excel report. See "Export Polyspace Analysis Results" on page 25-5. 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 `.xslx`, 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 25-5

Export Global Variable List

You can export the list of global variables in your code to a tab delimited text file or a MATLAB table. The text file or the table contains the same information as the **Variable Access** pane in the Polyspace user interface.

Using the text file, you can:

- Generate graphs or statistics about global variables. For instance, you can see the percentage of shared global variables that are not protected against concurrent access.
- Use the range information to create external constraints for global variables. For instance, you can report that your code is free of certain run-time errors only for the extracted range of global variables.

  You can also use the range to specify external constraints on subsequent verifications or verification of other modules. See "Specify External Constraints for Polyspace Analysis".

Export Variable List to Text File

You can export results from the user interface or command line.

User Interface	Command Line
1 Open your verification results. **2** Select **Reporting** > **Export** > **Export Variable Access**. **3** Select a location to save the text file and click **OK**.	Use appropriate options with the `polyspace-report-generator` command. The available options are: • `-generate-variable-access-file`: Specifies that a text file must be generated. • `-results-dir` *`folder_paths`*: Path to folder containing your analysis results. If you do not specify a folder path, the software uses analysis results from the current folder. To generate text files for multiple analyses, specify *`folder_paths`* as comma separated list with no spaces after the commas. For example: `C:\My_project\Module_1\results,C:\My_project\Module_2\Results` To merge the text files, use the `join` function. • `-set-language-english`: Use this option to generate English reports if the default report is in another language. The display language (Windows) or locale (Linux) of your operating system determines the default language in the report.

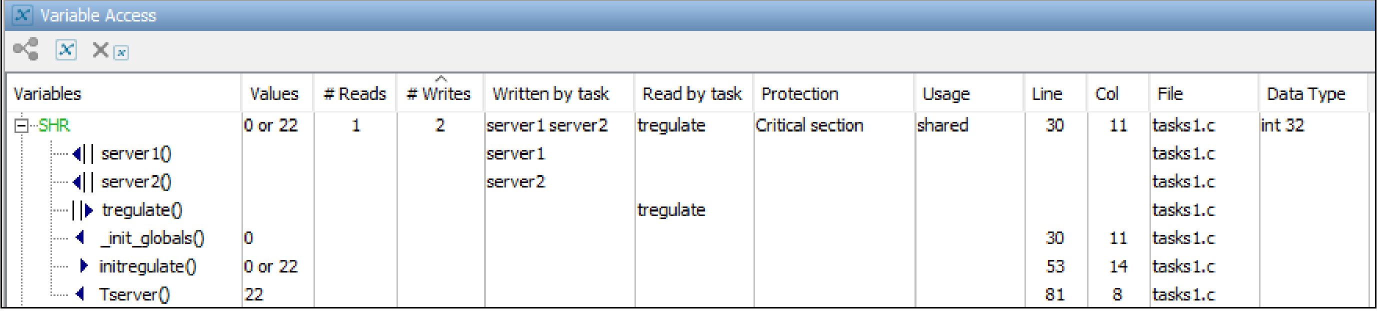
Export Variable List to MATLAB Table

Instead of a text file, you can read your Polyspace analysis results into a MATLAB table. See `variableAccess`.

View Exported Variable List

The text file or the table contains the result information available on the **Variable Access** pane in the user interface.

For instance, suppose the **Variable Access** pane shows a variable SHR with this information.



Variable Access

Variables	Values	# Reads	# Writes	Written by task	Read by task	Protection	Usage	Line	Col	File	Data Type
SHR	0 or 22	1	2	server1 server2	tregulate	Critical section	shared	30	11	tasks1.c	int 32
server1()				server1						tasks1.c	
server2()				server2						tasks1.c	
tregulate()					tregulate					tasks1.c	
_init_globals()	0							30	11	tasks1.c	
initregulate()	0 or 22							53	14	tasks1.c	
Tserver()	22							81	8	tasks1.c	

If you export this information to the tab-delimited text file and open the file in Excel, you see the same information. Some of the information is shown below.

Variables	Data Type	Access	Values	#Reads	#Writes	Written by task	Read by task	Protection	Line	Col	File	Function	Extension
SHR	int32	Aggregate	0 or 22	1	2	server1 server2	tregulate	Critical section	30	11	tasks1.c		c
SHR		Write	0						30	11	tasks1.c	_init_globals()	c
SHR		Write	22						81	8	tasks1.c	Tserver()	c
SHR		Read	0 or 22						53	14	tasks1.c	initregulate()	c

See also “Variable Access in Polyspace Desktop User Interface” on page 21-27.

Some differences in presentation between the **Variable Access** pane and the text file (or MATLAB table) are listed below.

- The **Access** column in the text file indicates whether the row shows information about the variable (**Aggregate**) or information about operations on the variable (**Write** or **Read**).
- The **Function** column in the text file shows the functions where the variable is read or written (and on the **Variable Access** pane).

- There are no rows corresponding to read and write operations from tasks (and on the **Variable Access** pane). This information is available in the **Written by task** and **Read by task** columns in the text file (`Tasks_Write` and `Tasks_Read` columns in the MATLAB table).
- The colors on the **Variable Access** pane are represented through the columns **Unreachable** and **Protected**:
  - If a shared variable is accessed in multiple tasks without a common protection, it is colored orange on the **Variable Access** pane. In the text file, the **Protected** column shows **Unprotected**.
  - If a shared variable is accessed in multiple tasks but with a common protection, it is colored green on the **Variable Access** pane. In the text file, the **Protected** column shows **Protected**.
  - If a shared variable is not accessed at all, it is colored gray on the **Variable Access** pane. In the text file, the **Unreachable** column shows **Is unreachable**.
- The **Potential** column in the text file shows read or write operations via pointers (or on the **Variable Access** pane). For operations via pointers, the column shows **Potential access**.

See Also

Related Examples

- “Export Polyspace Analysis Results”
- “Variable Access in Polyspace Desktop User Interface” on page 21-27

Visualize Code Prover Analysis Results in MATLAB

After 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.CodeProverResults` object associated with the results.

For instance, to read the demo results in the read-only subfolder `polyspace/examples/cxx/Code_Prover_Example/Module_1/CP_Result` of the MATLAB installation folder, copy the results to a writable folder and read them:

```
resPath = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Code_Prover_Example', 'Module_1','CP_Result');

userResPath = tempname;
copyfile(resPath,userResPath);

resObj = polyspace.CodeProverResults(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”.

Generate Graphs from Results and Include in Report

You can visualize the analysis results in the MATLAB table 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 red, gray and orange run-time checks by check type, 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', ...
    'Code_Prover_Example','Module_1','CP_Result');
userResPath = tempname;
copyfile(resPath,userResPath);

% Read results into a table.
resObj = polyspace.CodeProverResults(userResPath);
resTable = getResults(resObj);

% Keep results that are run-time checks and eliminate green checks.
matches = (resTable.Family == 'Run-time Check') &...
    (resTable.Color  ~= 'Green');
checkTable = resTable(matches, :);

% Create a pie chart showing distribution of checks.
```

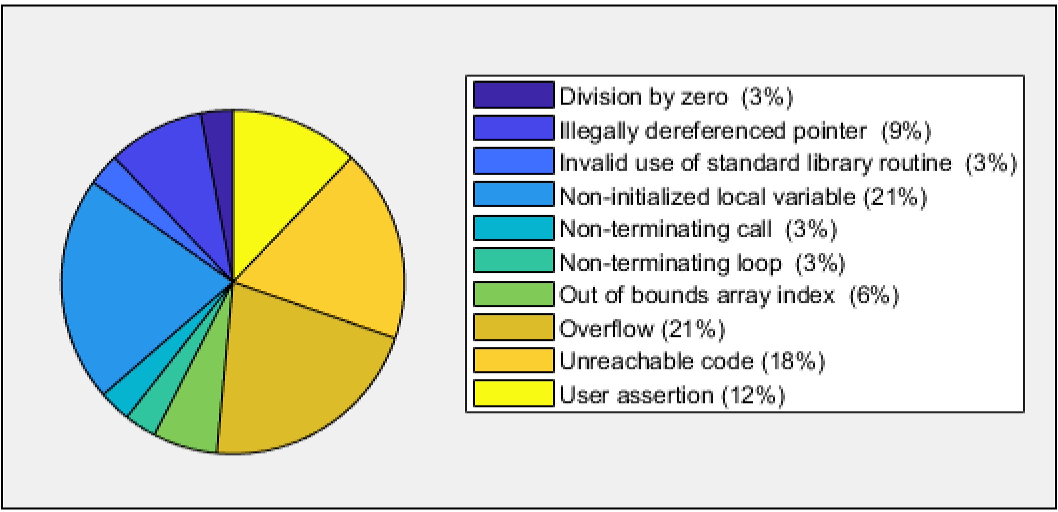
```
checkList = removecats(checkTable.Check);
pieChecks = pie(checkList);
labels = get(pieChecks(2:2:end),'String');
set(pieChecks(2:2:end),'String','');
legend(labels,'Location','bestoutside')

% 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,'Code Prover Run-Time Errors Graph'));
paragraphText = ['The following graph shows the distribution of ' ...
               'run-time errors 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,'Code Prover Run-Time Errors Details'));
paragraphText = ['The following table shows the run-time errors ' ...
               'in your code.'];
append(report, Paragraph(paragraphText));

% Add the table of run-time errors to the report.
reducedInfoTable = checkTable(:,{'File','Function','Check','Color',...
    '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.CodeProverResults`: Read Code Prover results into 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 checks by check type. The script also creates an HTML report that contains the graph and table of Polyspace checks.



See Also

Related Examples

- "Export Polyspace Analysis Results"
- "Export Polyspace Analysis Results to Excel by Using MATLAB Scripts"

Customize Existing Code Prover Report Template

In this example, you learn how to customize an existing report template to suit your requirements. A report template defines the content and formatting of reports generated from analysis results. 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 verification results using the template. See "Generate Reports from Polyspace Results".
- Make sure you have MATLAB Report Generator installed on your system.

In this example, you modify the **Developer** template that is available in Polyspace Code Prover.

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 **Developer** template.

1. Add paths to Polyspace-specific report components by pointing to subfolders of your Polyspace installation folder. At the MATLAB command prompt, enter:

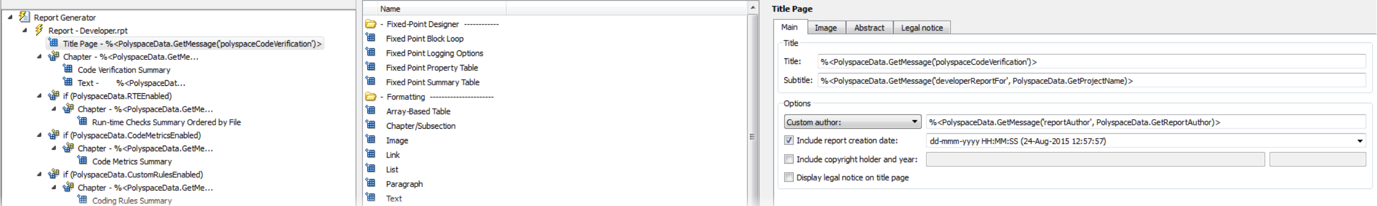
   ```
   addpath(fullfile(polyspaceroot, 'toolbox', 'polyspace', 'psrptgen', 'psrptgen'));
   addpath(fullfile(polyspaceroot, 'toolbox', 'polyspace', 'psrptgen', 'templates'));
   ```

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

   ```
   report
   ```
3. Open the **Developer** template in the Report Explorer interface.

   The **Developer** template is in *`polyspaceroot`*`/toolbox/polyspace/psrptgen/templates` 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 **Developer** template and their purpose are described below.

Component	Purpose
`Title Page`	Inserts title page in the beginning of report
`Chapter/Subsection`	Groups portions of report into sections with titles
`Code Verification Summary`	Inserts summary table of Polyspace analysis results
`Logical If`	Executes child components only if a condition is satisfied
`Run-time Checks Summary Ordered by File`	Inserts a table with Polyspace Code Prover checks 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 all 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 **Developer** 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.

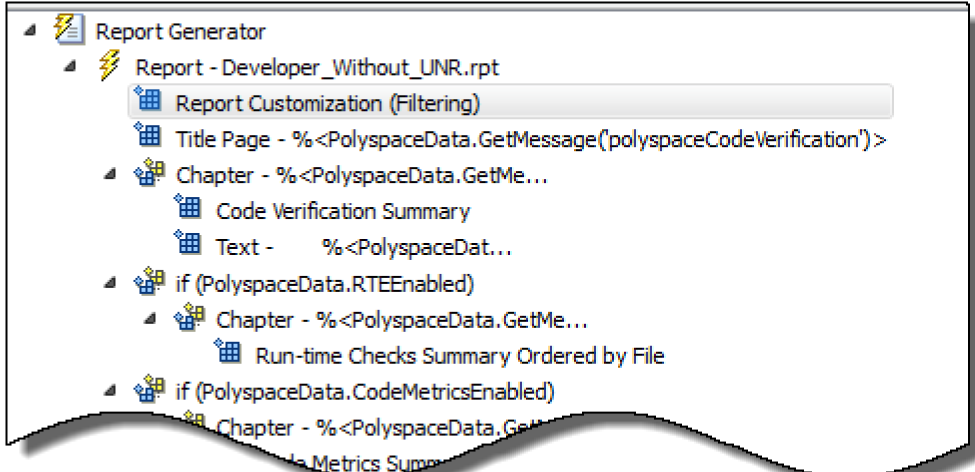
In this example, you add a component to the **Developer** template that filters `Unreachable code` checks from a report generated using the template.

1. Open the **Developer** template in the Report Explorer interface and save it elsewhere with a different name, for instance, **Developer_without_UNR**.
2. Add a new global component that filters **Unreachable code** checks from the **Developer_without_UNR** template. The component is global because it applies to the full report and not one chapter of the report.

   To perform this action:

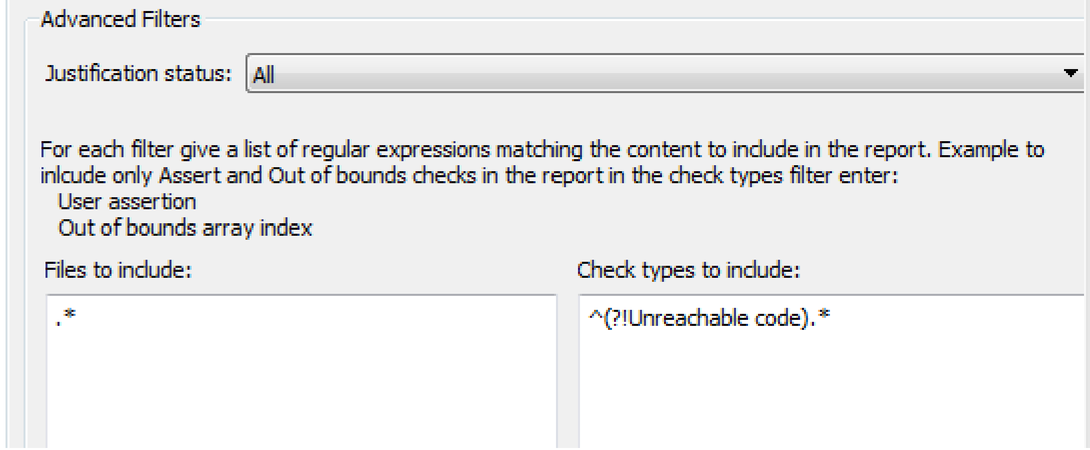
   a. Drag the component `Report Customization (Filtering)` located under **Polyspace** in the middle pane in the middle pane and place it above the **Title Page** component. The

positioning of the component ensures that the filters apply to the full report and not one chapter of the report.



b Select the **Report Customization (Filtering)** component. On the right pane, you can set the properties of this component. By default, the properties are set such that all results are included in the report.

To exclude **Unreachable code** checks, under the **Advanced Filters** group, enter `^(?!Unreachable code).*` in the **Check types to include** field.



You can enter MATLAB regular expressions in this field. 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 `^(?!Unreachable code).*` indicates that Polyspace result names beginning with `Unreachable code` must be excluded from the report. See “Regular Expressions” and “Complete List of Polyspace Code Prover Results”.

You can toggle between activating and deactivating this component. Right-click the component and select **Activate/Deactivate Component**.

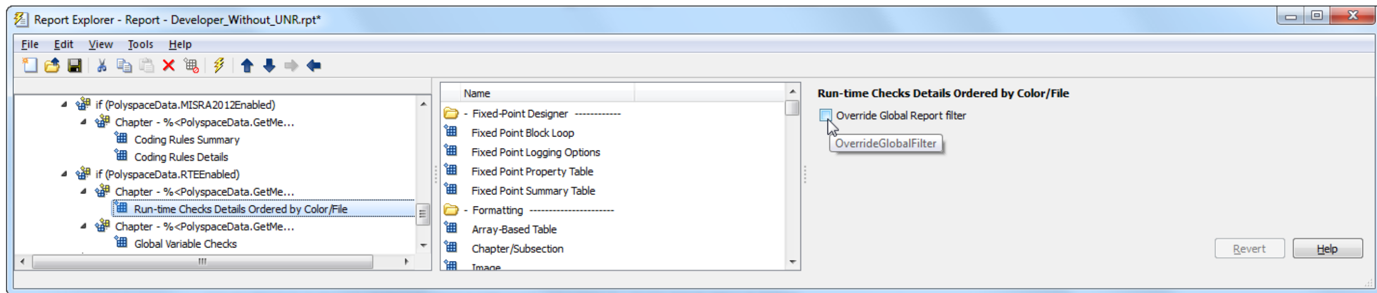
3. Change an existing chapter-specific component so that it does not override the global filter you applied in the previous step. If you prevent the overriding, the chapter-specific component follows the filtering specifications in the global component.

   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 Code Prover.

      The right pane shows the properties of this component.

   b. Clear the **Override Global Report** filter box.



   Save the **Developer_without_UNR** template after making your changes.

4. In the Polyspace user interface, create a report using both the **Developer** and **Developer_without_UNR** template from results containing **Unreachable code** checks. Compare the two reports.

   For instance:

   a. Open **Help > Examples > Code_Prover_Example.psprj**.

      The demo result contains **Unreachable code** checks.

   b. Create a pdf report using the **Developer** template.

      In the report, open **Chapter 5. Polyspace Run-Time Checks Results** (in your version of the product, the chapter number might be different). *You can see gray **Unreachable code** checks.* Close the report.

   c. Create a pdf report using the **Developer_without_UNR** template. In the Run Report window, use the **Browse** button to add the **Developer_without_UNR** template to the existing template list.

      In the report, open **Chapter 6. Polyspace Run-Time Checks Results** (in your version of the product, the chapter number might be different). *You do not see gray **Unreachable code** checks.*

Note After you add the template to the existing list of templates, before generating the report, make sure to select the newly added template.

Further Exploration

Modify the **Developer** template such that the file `initialisations.c` is excluded from a report generated using the template. Generate a report from **Code_Prover_Example** results using your modified template and verify that the file `initialisations.c` is excluded from the report.

Hint: The regular expression you must use is `^(?!.*initialisations.c).*`

For more examples, see "Sample Report Template Customizations" on page 25-22.

See Also

`Bug Finder and Code Prover report (-report-template)` | `Generate report` | `Output format (-report-output-format)`

Related Examples

- "Generate Reports from Polyspace Results"
- "Sample Report Template Customizations" on page 25-22

Sample Report Template Customizations

A report template defines the content and formatting of reports generated from analysis results. If an existing template does not suit your requirements, you can change certain aspects of the template.

This topic shows some customizations you can do to a Polyspace report template, with brief steps. For a more detailed tutorial, see “Customize Existing Code Prover Report Template” on page 25-17.

To customize a template:

1. Open MATLAB Report Generator. At the MATLAB command prompt, enter:

   ```
   report
   ```
2. Open an existing template.

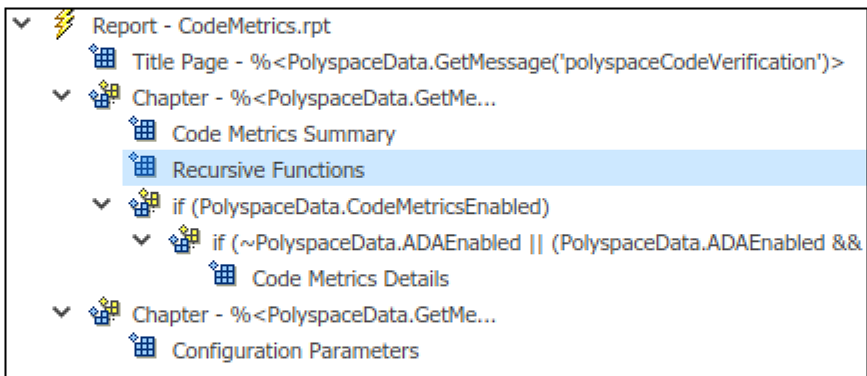
   The templates are located in `*polyspaceroot*/toolbox/polyspace/psrptgen/templates`. *`polyspaceroot`* is the Polyspace installation folder.
3. Add, remove, or modify components of the template.

   For a full list of Polyspace-specific components, see “Generate Reports”.

Add List of Recursive Functions

Suppose that you want to report all recursive functions detected in your source code.

Start from the **CodeMetrics** template. In the chapter on code metrics, add the component `Recursive Functions`.



When you generate a report by using the modified template, you see a table with the list of recursive functions.

Show Red Run-Time Checks Only

Suppose that you want to report an overview of all run-time checks and details for red checks only.

Start from the **Developer** template. Remove all chapters, except the ones containing these components:

- `Code Verification Summary`

- `Run-time Checks Summary Ordered by File`
- `Run-time Checks Details Ordered by Color/File`. Modify this component so that it shows red checks only.

  Select the component. On the right pane, in the group **Categories To Include**, clear all boxes other than **Red Checks**.
- Appendix components: `Configuration Parameters` and `Acronym Definitions`.

Report - Developer.rpt*
Title Page - %<PolyspaceData.GetMessage('polyspaceCodeVerification')>
Chapter - %<PolyspaceData.GetMe...
Code Verification Summary
Text - %<PolyspaceDat...
if (PolyspaceData.RTEEnabled)
Chapter - %<PolyspaceData.GetMe...
Run-time Checks Summary Ordered by File
if (PolyspaceData.RTEEnabled)
Chapter - %<PolyspaceData.GetMe...
Run-time Checks Details Ordered by Color/File
Chapter - %<PolyspaceData.GetMe...
Configuration Parameters
Chapter - %<PolyspaceData.GetMe...
Acronym Definitions

When you generate a report by using the modified template, you see an overview of checks, a chapter with details for red checks only, and the appendix.

Show Non-Justified Run-Time Checks Only

Suppose that you want to report only the checks that you have not justified. You justify a check when you assign one of these statuses:

- `Justified`
- `No action planned`
- `Not a defect`

Add the component `Report Customization (Filtering)` above the first chapter. Modify the component so that the following chapters show non-justified checks only.

Title Page - %<PolyspaceData.GetMessage('polyspaceCodeVerification')>
Report Customization (Filtering)
Chapter - %<PolyspaceData.GetMe...
Code Verification Summary
Text - %<PolyspaceDat...
if (PolyspaceData.RTEEnabled)
Chapter - %<PolyspaceData.GetMe...
Run-time Checks Summary Ordered by File

Select the component. On the right pane, in the group **Advanced Filters**, from the **Justification Status** list, select `Un-justified`.

When you generate a report by using the modified template, you see only the non-justified run-time checks.

Add Chapter for Functional Design Errors

Suppose that you implement functional design testing using `assert` statements in your code. For instance, to test if the output of a function `out` is within a range `[MIN,MAX]`, your code uses the statement:

```
assert(MIN <= out && out <= MAX);
```

Polyspace runs the check `User assertion` to determine if the `assert` condition fails. Suppose that you want to report these checks in a separate chapter because they are different from the other run-time error checks.

Start from the **Developer** template. Make a copy of the chapter containing the component `Run-time Checks Details Ordered by Color/File`.

if (PolyspaceData.RTEEnabled)
Chapter - %<PolyspaceData.GetMe...
Run-time Checks Details Ordered by Color/File
Chapter - %<PolyspaceData.GetMe...
Run-time Checks Details Ordered by Color/File

Rename each of the two chapter titles so that you can distinguish between them. In each chapter, modify the component **Run-time Checks Details Ordered by Color/File** as follows:

- In one chapter, exclude **User assertion** checks. Select the component. On the right pane, in the group **Advanced Filters**, for **Check types to include**, enter:

  ```
  ^(?!User assertion).*
  ```
- In the other chapter, include **User assertion** checks. Select the component. On the right pane, in the group **Advanced Filters**, for **Check types to include**, enter:

  ```
  User assertion
  ```

  Clear the boxes for grey checks, because the **User assertion** checks cannot be grey.

When you generate a report by using the modified template, you see two copies of the chapter on run-time checks. The first chapter contains all checks other than **User assertion** checks and the second chapter contains **User assertion** checks only.

See Also

Related Examples

- "Customize Existing Code Prover Report Template" on page 25-17

Generate Report Containing MISRA C:2012 Violations, Code Metrics, and Runtime Check Results

To obtain a report that contains the Code Prover results, all code metrics, and MISRA C:2012 violations, run the command `polyspace-report-generator`. Generate a combined report containing these results:

- MISRA C:2012. See "MISRA C:2012 Directives and Rules".
- Code metrics. See "Code Metrics".
- Stack usage metrics. See Stack Usage Metrics.
- Run-time checks. See "Run-Time Checks".

For more information, see `polyspace-report-generator`.

Code Prover does not check for violations of coding standards, such as MISRA C:2012. Use Bug Finder to check for coding rule violations. Code Prover computes the stack usage code metrics only. The other code metrics are calculated by Bug Finder.

Prerequisite

- Before generating the report, confirm that your installed Polyspace version is R2021b or later.
- To use the C source file used in this example, navigate to *`polyspaceroot`*`\polyspace\examples\cxx\Bug_Finder_Example\sources\numerical.c`. Substitute *`polyspaceroot`* with your Polyspace installation path, for instance, `C:\Program Files\Polyspace\R2023a`. See also "Installation Folder".

Obtain Code Metrics and Coding Rules Results by Using Bug Finder

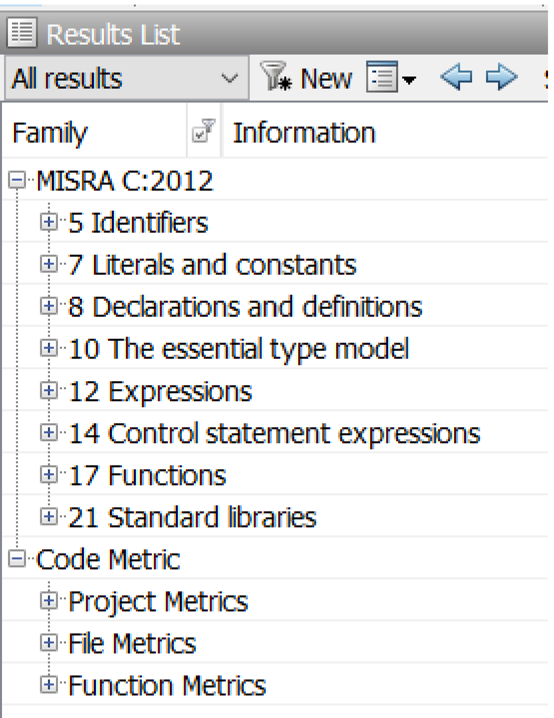
Run a Bug Finder analysis and store the results.

1. Run Bug Finder on `numerical.c` to check for coding rule standard violations, and then store the results in a folder named `BFResults`. For instance, at the command line, enter:

   ```
   polyspace-bug-finder
   -sources "polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources\numerical.c"
   -results-dir BFResults -code-metrics -misra3 all-rules -lang c -checkers none
   ```

   The command runs Bug Finder on `numerical.c` and stores the results in a folder named BFResults. Bug Finder checks for violations of MISRA C:2012 coding rules and computes the code metrics.

   In the next steps, Code Prover performs an exhaustive check for run-time errors and other issues. In such a case, activating the Bug Finder defects might be redundant. Polyspace Report Generator does not support putting Code Prover run-time errors and Bug Finder defects in the same report. The defect checkers are deactivated in this step.
2. Navigate to *`Current Folder`*`\BFResults` and open the file `ps_results.psbf`. Verify that the results contain MISRA C;2012 violations and code metrics results.



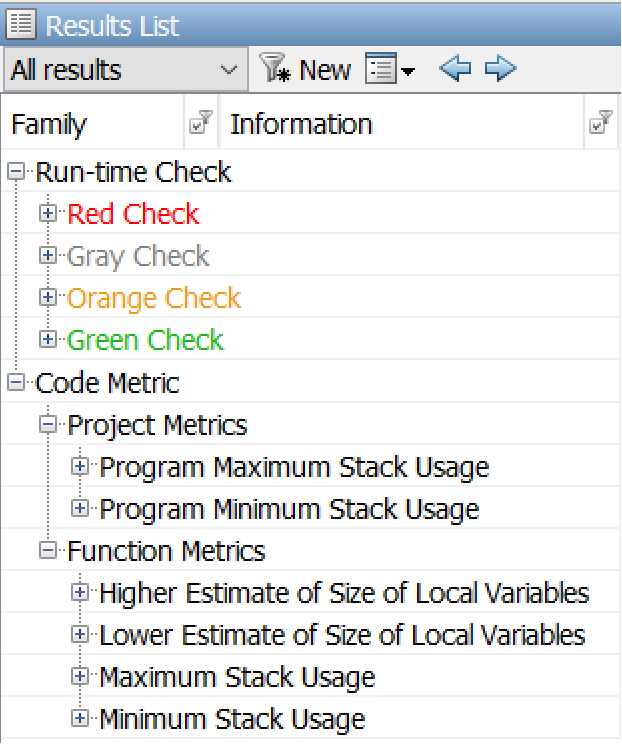
Obtain Run Time Check and Stack Usage Results by Using Code Prover

Run a Code Prover analysis and store the results.

1. Run Code Prover on `numerical.c` to check for run-time issues and to calculate stack usage metrics. Store the results in a folder named `CPResults`. For instance, at the command line, enter:

   ```
   polyspace-code-prover
   -sources “polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources\numerical.c”
   -results-dir CPResults -stack-usage -lang c -main-generator
   ```

   The command runs Code Prover on `numerical.c` and stores the results in a folder named `CPResults`. Because `numerical.c` does not have a `main()` function, specify the option `Verify module or library (-main-generator)`.
2. Navigate to *`Current Folder`*`\CPResults` and open the file `ps_results.pscp`. Verify that the results contain run-time checks and stack usage results.



Generate a Combined Report

After generating the Bug Finder and Code Prover results, summarize the results into a single report. To generate the report, specify an appropriate template and the results folders as inputs to the command `polyspace-report-generator`.

1. Before executing the report generation command, select an appropriate template. For this example, the report must contain a chapter each for code metrics, Code Prover runtime checks, and MISRA C:2012 violations. The template `Developer.rpt` accommodates all of these results.

   To include additional chapters about coding standards such as AUTOSAR C++14 or CERT C/C++, modify the existing templates. See “Customize Existing Code Prover Report Template” on page 25-17.
2. Start the report generation by using the command `polyspace-report-generator`. Specify the template `Developer.rpt` as the input to `-template`. Provide the folders containing the Bug Finder and Code Prover results as inputs to `-results-dir`. At the command line, enter:

```
polyspace-report-generator
-template "polyspaceroot\toolbox\polyspace\psrptgen\templates\Developer.rpt"
-results-dir “CPResults”,”BFResults” -output-name combined_report -format PDF
```

The file `combined_report.pdf` is saved in your current folder. Open the file and verify that the report contains code metrics, Code Prover run-time checks, and MISRA C:2012 violations.

Chapter 1. Polyspace Code Verification Summary
Chapter 2. Polyspace Run-Time Checks Statistics
Chapter 3. Code Metrics
Chapter 4. MISRA C:2012 Guidelines
Chapter 5. Polyspace Run-Time Checks Results
Chapter 6. Global Variables
Chapter 7. Appendix 1 - Configuration Settings
Chapter 8. Appendix 2 - Definitions

See Also

`polyspace-report-generator` | `polyspace-code-prover`

Related Examples

- “Customize Existing Code Prover Report Template” on page 25-17
- “Export Polyspace Analysis Results”
- “Generate Reports from Polyspace Results”
- “Migrate Code Prover Workflows for Checking Coding Standards and Code Metrics to Bug Finder” on page 16-52

Review Results on Web Browser

Interpret Polyspace Code Prover Access Results

- “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2
- “Dashboard in Polyspace Access Web Interface” on page 26-7
- “Code Metrics Dashboard in Polyspace Access Web Interface” on page 26-9
- “Quality Objectives Dashboard in Polyspace Access” on page 26-12
- “Results List in Polyspace Access Web Interface” on page 26-17
- “Source Code in Polyspace Access Web Interface” on page 26-19
- “Result Details in Polyspace Access Web Interface” on page 26-24
- “Call Hierarchy in Polyspace Access Web Interface” on page 26-26
- “Configuration Settings in Polyspace Access Web Interface” on page 26-28
- “Global Variables in Polyspace Access Web Interface” on page 26-31
- “Review History in Polyspace Access Web Interface” on page 26-36
- “Create Bug Tracking Tool Tickets from the Polyspace Access Web Interface” on page 26-38

Interpret Code Prover Results in Polyspace Access Web Interface

This topic shows how to review Code Prover results in the Polyspace Access web interface. For a similar workflow in the user interface of the Polyspace desktop products, see “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2.

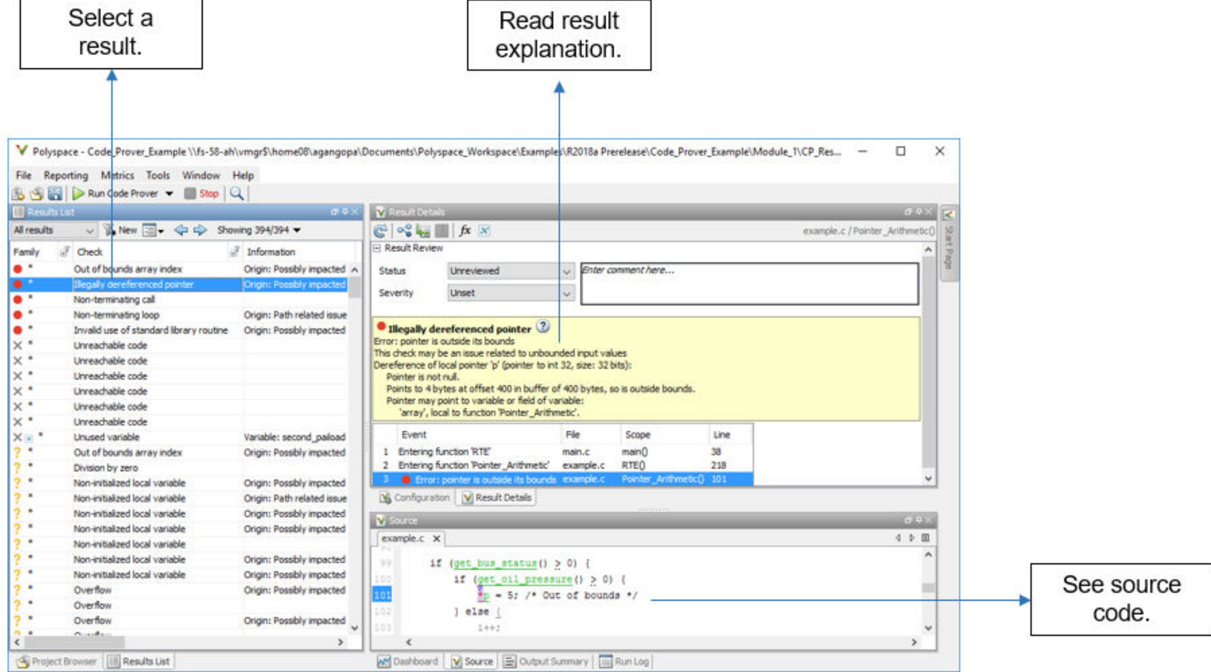
When you open the results of a Polyspace Code Prover analysis, you see a list on the **Results List** pane. The list consists of run-time checks, coding rule violations, code metrics and global variable usage.

You can first narrow down the focus of your review:

- Use filters in the toolstrip to narrow down the list. For instance, you can focus on the high-impact defects.
- Click the a column header in the **Results List** to sort the list according to the content of that column. For instance you can sort by **Group** or by **File**.

  Because the results of a Code Prover run-time check are dependent on the results of previous checks, it helps to go through run-time checks from the beginning to the end of a function.

See also “Filter and Sort Results in Polyspace Access Web Interface” on page 28-8. 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

Interpret Message

The first step is to understand what the issue is. Read the message on the **Result Details** pane and the related line of code on the **Source** pane.

At this point, you might be ready to decide whether to fix the issue.

Out of bounds array index
Warning: array index may be outside bounds : [0..126]
This check may be an issue related to unbounded input values
Local variable main.PORT_A that is declared volatile in main.c line 48 may lead to imprecise check
Local variable main.PORT_B that is declared volatile in main.c line 49 may lead to imprecise check
array size: 127
array index value: [0 .. 555]

The message consists of several parts:

- Check color and icon: See “Code Prover Result and Source Code Colors” on page 32-2. In case of checks for run-time errors:
  - : Red indicates a definite error.
  - : Orange indicates a possible error.
  - : Gray indicates unreachable code.
  - : Green indicates that a specific error cannot happen.
- Description of the run-time check.

  In the preceding example, the check determines if an array index goes outside the array bounds.
- Values relevant to the run-time check.

  In the example, the message states the array size (127), the array bounds (0..126), and the range of values that the array index variable can take at that point in the code (0..555).
- Relevant sources of imprecision (for orange checks).

  In the example, the message states that two volatile variables might be responsible for the check.

See Variable Ranges in Source Code Tooltips

On the **Source** pane, variables and operations with tooltips are underlined.

```
output_v1 = s8_ret + 15;
```

In this example, tooltips appear on:

- `s8_ret`: You see its data type and range of values before the + operation.

  If a data type conversion occurs during the + operation, you also see this conversion in the tooltip.
- +: You see the value of the left and right operand, and the result.

- =: You see any data type conversion that occurs during the assignment and the result.

Get Additional Help

Sometimes, you need additional help for certain results. To open a help page for the selected result, click the icon. See code examples that illustrate the result.

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 possibly a previous `if` or `while` condition that is always false.

Navigate in Source Code

Sometimes, the **Result Details** pane shows one sequence of events that leads to the result. However, in most situations, you have to find your own navigation pathways through the code. Using tooltips on variables, follow the propagation of variable ranges as you navigate through the code.

```
int func (int var) { /* Initial range of var */
     …
     var -= get ();  /* New range of var */
     …
     set(&var);     /* New range of var */
}
```

Use these quick navigation pathways in the user interface:

- Search for all references to a variable and browse through them.

  Right-click the variable name on the **Source Code** pane and select **Search For All References**. Alternatively, double-click the variable. These options perform more than a string match. The options show only instances of a specific variable and not other variables with the same name in other scopes.
- Navigate from a function call to its definition.

  Right-click the function name on the **Source Code** pane. Select **Go To Definition**.
- Navigate from a function to its callers and callees.

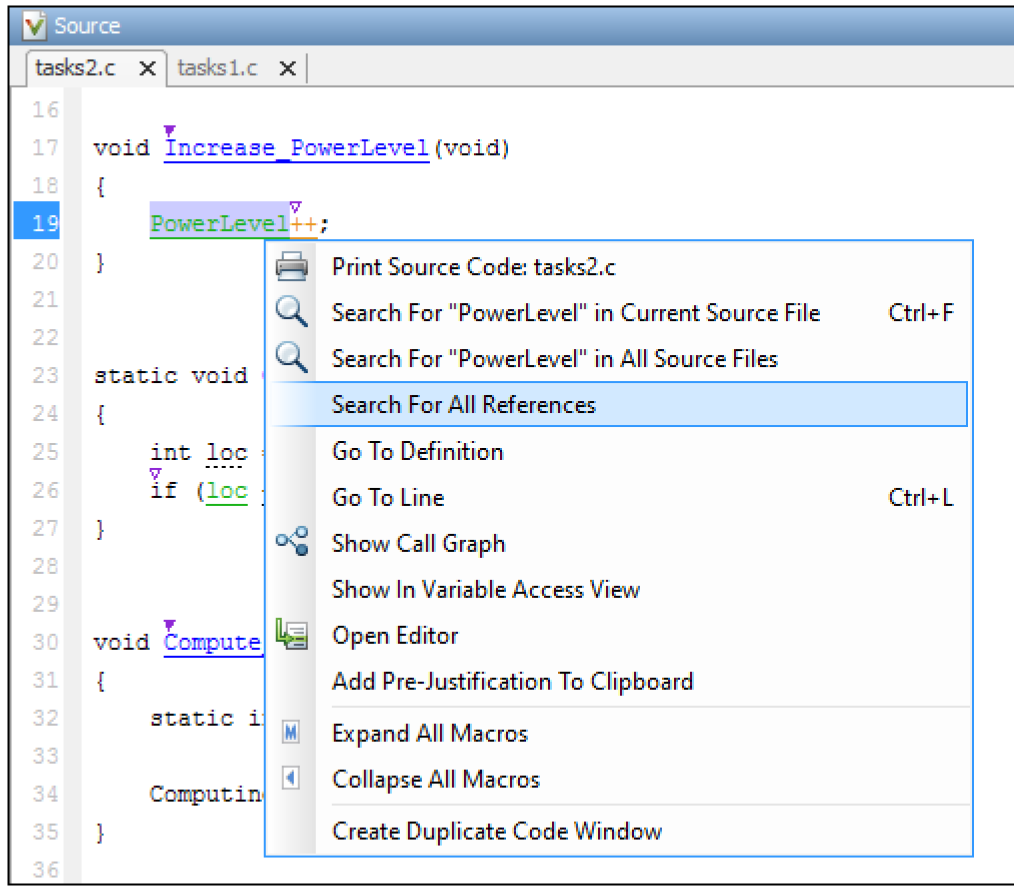
  Click the icon on the **Result Details** pane. You see the function containing the result, with its callers and callees. Click a caller or callee name to navigate to the call site. Double-click a name to navigate to the definition.

  Alternatively, click the icon to see a graphical representation of the call sequence leading to the result. To navigate to functions in this sequence, click through nodes in the graph.
- Navigate from a function call or loop keyword to an error in the function or loop body.

  If the error occurs only in a specific function call or specific loop iteration, the function call or loop iteration is highlighted red. Right-click the red function call or loop keyword. Select **Go To Cause** if the option is available.

- Navigate across all instances of a global variable.

  Click the icon on the **Result Details** pane. See all global variables in the result and read/write operations on them.



Before you begin navigating through pathways in your code, determine what you are looking for and choose the appropriate navigation tool. For instance:

- To investigate a **Non-initialized variable** check, you might want to make sure that the variable is not initialized at all. Look for previous instances of the variable and see if it is initialized.
- 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.

For other examples of what to look for, see “Reviewing Code Prover Run-Time Checks” on page 32-7. After you navigate away from the current result, use the icon on the **Result Details** pane to return to that result.

If you click a source code token containing a result, the previous result selection on the **Results List** and 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 in the **Source Code** pane and select **Create Duplicate Code Window**. Right-click 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 in the original file window. After the investigation is complete, close the duplicate window.

See Also

More About

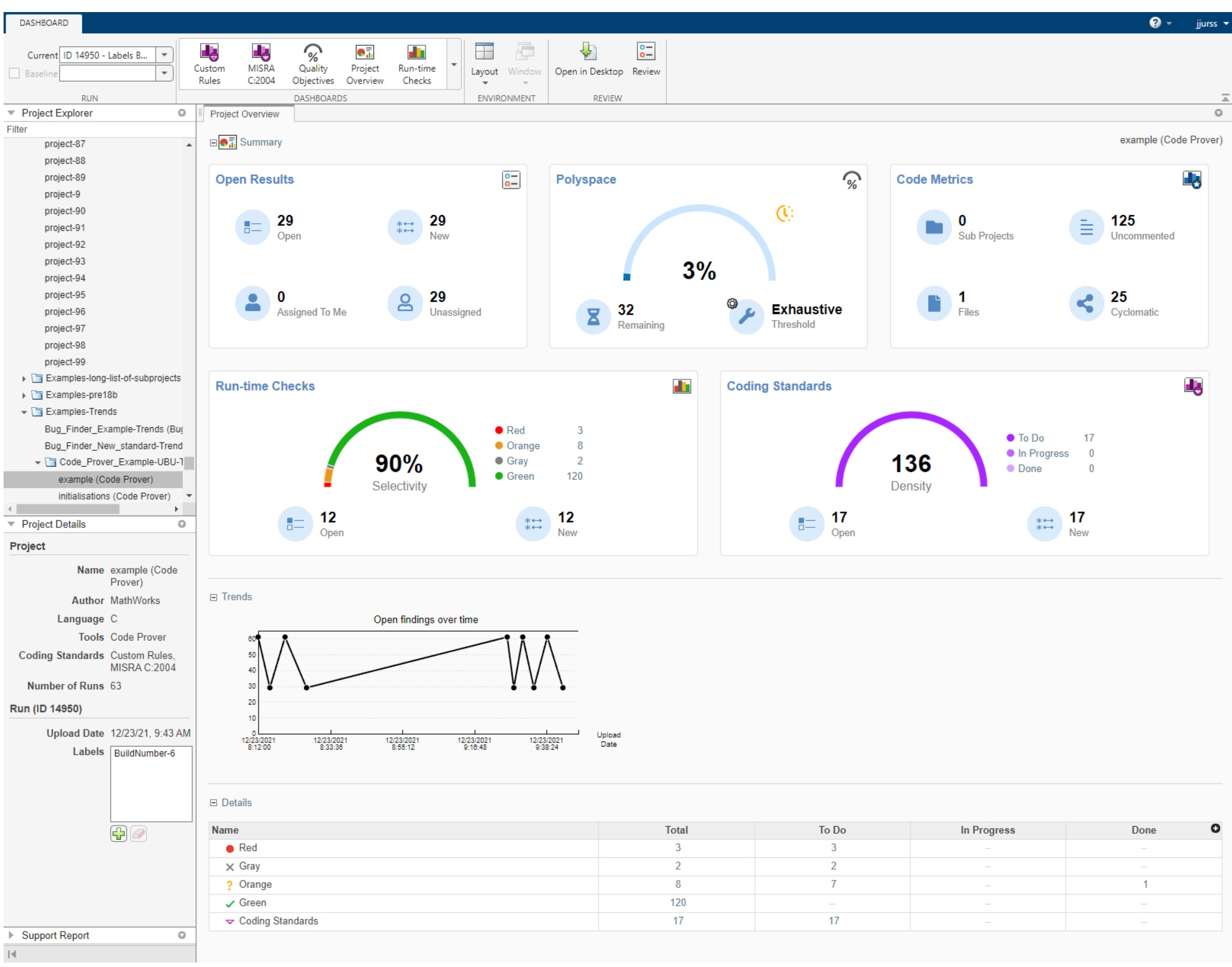
- "Filter and Sort Results in Polyspace Access Web Interface" on page 28-8
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" on page 27-2

Dashboard in Polyspace Access Web Interface

This topic focuses on the Polyspace Access web interface. To learn about the equivalent pane in the Polyspace desktop user interface, see “Dashboard in Polyspace Desktop User Interface” on page 21-7.

The **DASHBOARD** perspective provides an overview of the analysis results in graphical format, with clickable fields that let you drill down into your findings by project, file, or category.

When you upload an analysis run to the Polyspace Access database, the **DASHBOARD** updates to display the statistics for the latest run.



On the **Project Overview** dashboard, you see statistics for the currently selected project. When you select a folder in the **Project Explorer**, you see an aggregate of statistics for all the projects under that folder. The aggregate does not include the statistics of projects for which you do not have a role of **Administrator**, **Owner**, or **Contributor**.

In the **Summary** section of the **Project Overview** dashboard, cards display information about open issues, code metrics, quality objectives, and the different families of findings.

- The **Run-time Check** card (Code Prover) shows a distribution of findings as red, orange, gray, and green. The card also shows the **Selectivity**, the number of green checks as a percentage of all detected run-time checks.
- **Defects** and **Coding Rules** cards show a distribution of findings as:

  - **To Do** — Findings with a status of `Unreviewed` that need to be addressed with a fix or a justification.
  - **In Progress** — Findings with a status of `To fix`, `To investigate`, or `Other` that need to be addressed with a fix or a justification.
  - **Done** — Findings with a status of `Justified`, `No action planned`, or `Not a defect`.

  The card also shows the **Density**, the number of **To Do** and **In Progress** defects or coding standard violations per one thousand lines of code without comments. To view the density you must enable **Code Metrics** in your analysis.

Note Green run-time checks, green shared variables, non-shared variables, and code metrics do not need to be addressed or justified. These findings do not count toward the number of findings that are **To Do**, **In Progress**, and **Done**.

To see a more in-depth overview for a family of findings, open additional dashboards by clicking the corresponding card title in the **Project Overview** dashboard or by using the **DASHBOARDS** section of the toolstrip.

In the additional dashboards:

- The **Summary** section displays project statistics for that family of findings such as current progress of results review or code coverage information.
- The **Details** section displays a table that allows you to drill down into the findings by category or by file. If you select a folder that contains multiple projects, you see a categorization by project instead of by file.

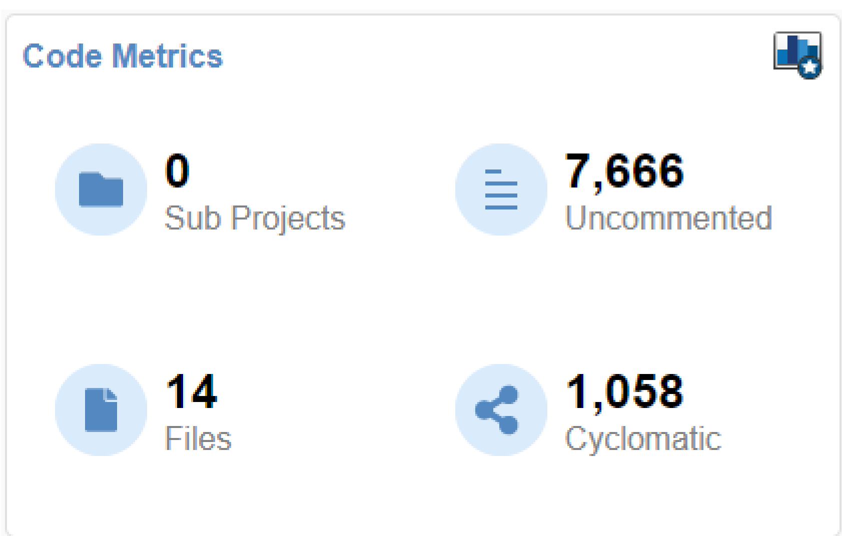
You can also perform these actions on the **DASHBOARD** perspective:

- View statistics for a previous run or compare a current run to a previous run. See “Compare Results in Polyspace Access Project to Previous Runs and View Trends” on page 28-23.
- Click elements on the graphs or tables to filter results from the **Results List** pane. See “Filter and Sort Results in Polyspace Access Web Interface” on page 28-8.
- Define and set quality objective levels. See “Quality Objectives Dashboard in Polyspace Access” on page 26-12.
- Manage projects and user authorizations. See “Manage Permissions and View Project Trends in Polyspace Access Web Interface” on page 28-2.
- Open the current project findings in the Polyspace desktop interface.

Code Metrics Dashboard in Polyspace Access Web Interface

To view the code complexity metrics that Polyspace computes, use the **Code Metrics** dashboard. See "Code Metrics".

Polyspace computes the code complexity metrics during an analysis only when you use the option `Calculate code metrics (-code-metrics)`.



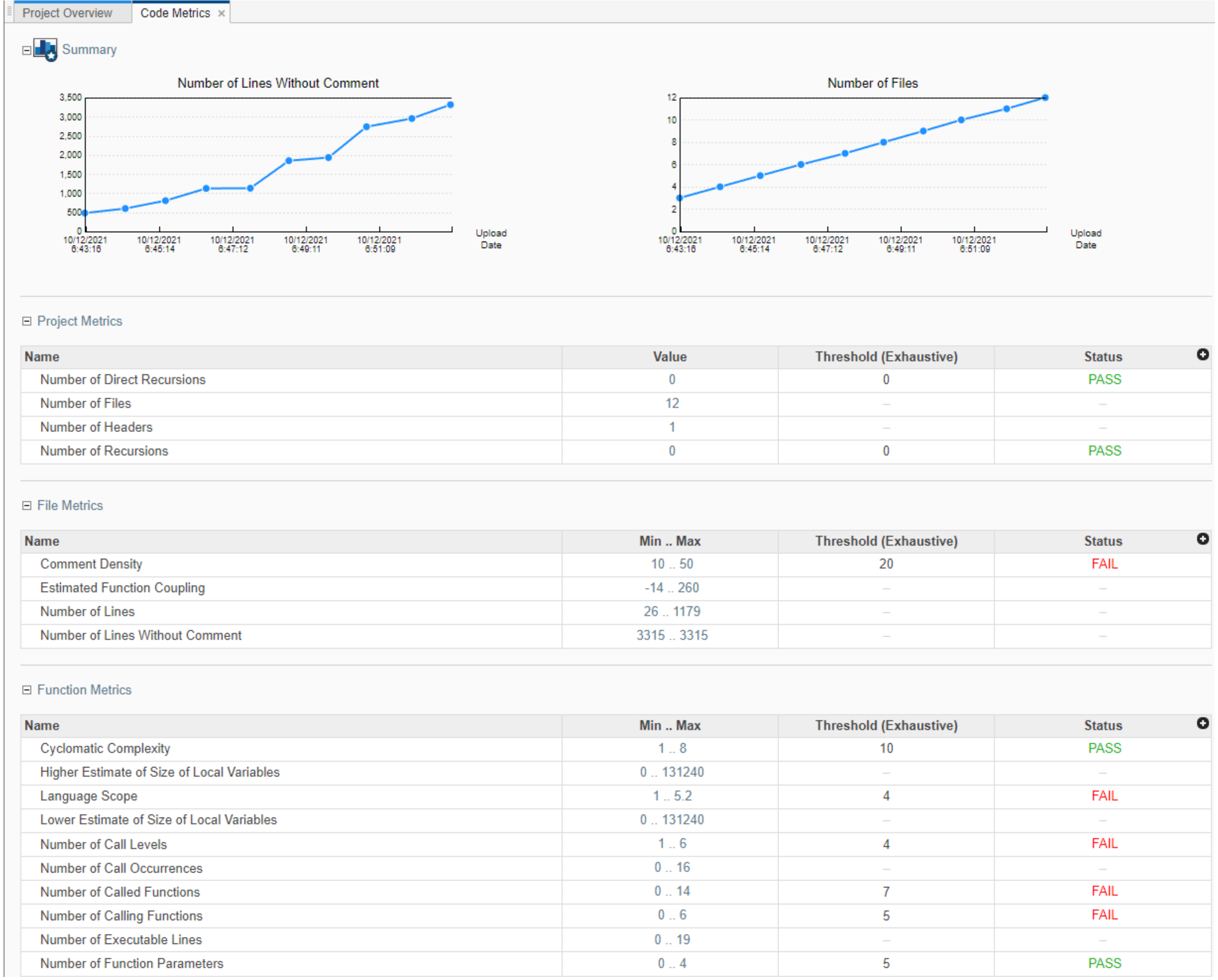
In the **Project Explorer**, select a project. Use the **Code Metrics** card in the **Project Overview** dashboard to get a quick overview of these code metrics:

- `Number of Files` (**Files**)
- `Number of Lines Without Comment` (**Uncommented**)
- `Cyclomatic Complexity` (**Cyclomatic**)

If you select a folder in the **Project Explorer**, the **Code Metrics** card shows:

- The number of **Sub Projects** in that folder. This number includes only subprojects that are directly (one level) below the top level folder.
- An aggregate of the other metrics on the card for all the subprojects at any level for which you are a **Contributor**, an **Owner**, or an **Administrator**.

To open the **Code Metrics** dashboard, click the **Code Metrics** icon in the **DASHBOARD** section of the toolstrip. Or, click **Code Metrics** on the card in the **Project Overview** dashboard.



In the **Summary** section, you see trend charts of the **Number of lines Without Comment** and **Number of Files** for the project.

The other sections of the dashboard display tables with the computed value or range of the different project, file, and function metrics. When applicable, the table shows the predefined threshold and pass/fail status for the corresponding code metric. For a list of code complexity metrics thresholds, see "HIS Code Complexity Metrics" on page 16-50. If you select a folder in the **Project Explorer**, the tables in the **Code Metrics** dashboard do not show the threshold or pass/fail status. The value or range of the metrics are aggregate of all subprojects in the selected folder. To drill down to a project from this aggregate view, expand a table row and click the project name.

To improve your code quality, use the pass/fail status to identify and lower metrics values that exceeds a threshold.

For instance, if the **Number of Called Functions** range exceeds the predefined threshold:

1. Click **FAIL** in the **Status** column or click the range in the **Min..Max** column to open the **Results List** filtered to the **Number of Called Functions** metric
2. Review the results that exceed the metric threshold. If several of those functions are always called together, you can write one function that fuses the bodies of those functions. Call that one function instead of the group of functions that are called together.

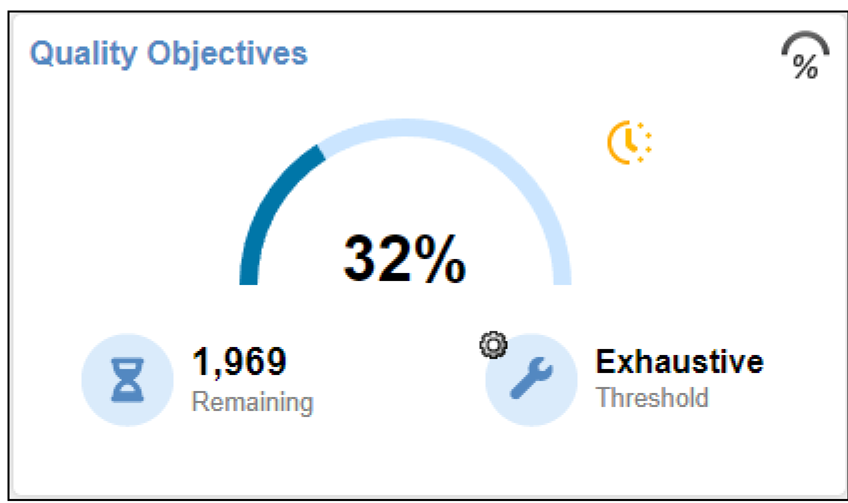
Quality Objectives Dashboard in Polyspace Access

To monitor the quality of your code against predefined on page 31-2 software quality thresholds or user-defined thresholds, use the **Quality Objectives** dashboard. You can use the thresholds as pass or fail criteria during the various stages of your project. From the dashboard, you can:

- Apply the default on page 31-2 Polyspace Access quality objectives or create custom objectives that suit your requirements. See "Customize Software Quality Objectives" on page 26-14.
- view a snapshot of your code quality against all levels of the currently applied quality objectives definition.

To manage the thresholds that you assign to projects, see "Manage Software Quality Objectives in Polyspace Access" on page 28-15.

Monitor Code Quality Against Software Quality Objectives



In the **Project Overview** dashboard, use the **Quality Objectives** card to get a quick overview of your progress in achieving a quality objective threshold. The card shows:

- The percentage of findings already addressed to achieve the selected threshold.
- One of these labels:
  - (pass) — All findings for this threshold have been addressed.
  - (in progress) — Some findings for this threshold are still open. A finding is open if it has a review status of `Unreviewed`, `To fix`, `To investigate`, or `Other`.
  - (incomplete) — Some checkers required for this threshold were not activated in the analysis. For instance, if a threshold requires that you address all Polyspace Bug Finder defects, but the analysis includes only **Numerical** defects, the level is incomplete, even if you address all findings. To see a list of checkers you must activate, click .

Note This label applies only to SQO thresholds 1 through 6. If you select the **Exhaustive** software quality threshold and you address all the findings, the threshold is labeled as (pass) even if all required checkers for this threshold were not activated in the analysis.

- (not computed) — No quality objective results were computed.

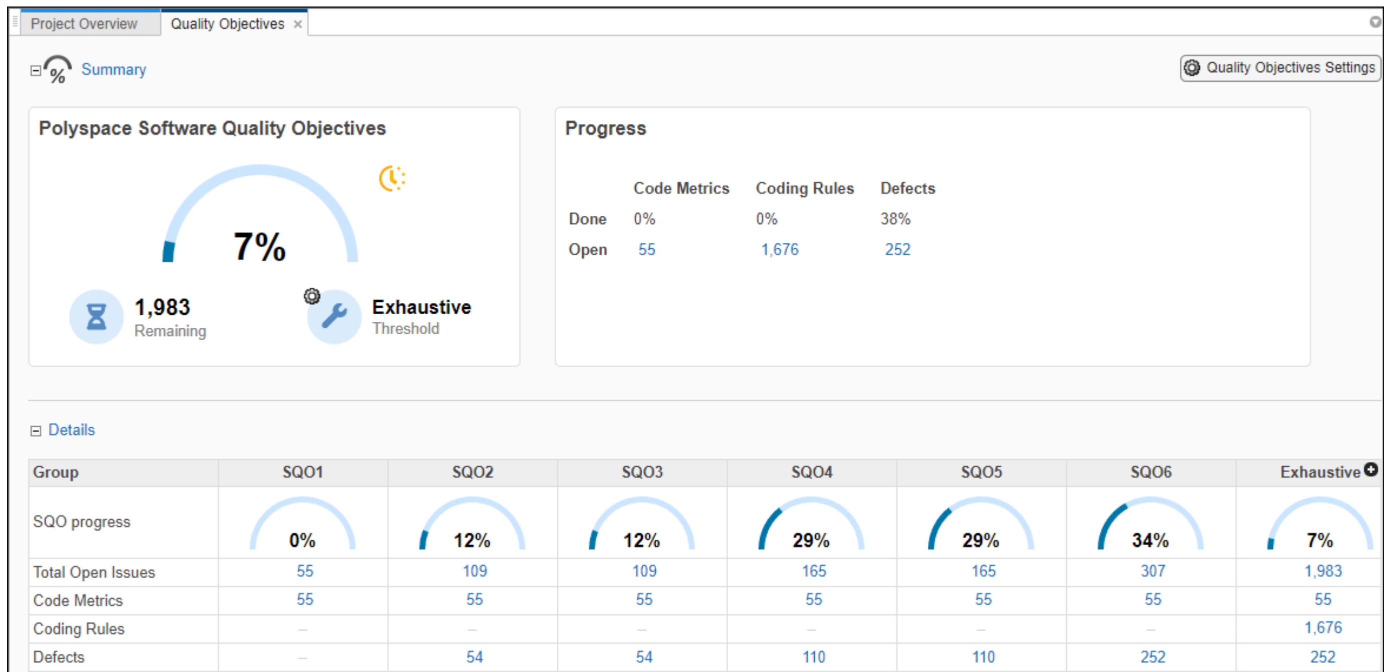
  After you apply a new quality objectives definition to a project, you see the not computed label until you upload a new run to the project.

- The assigned **Threshold**. To select a different threshold or quality objectives definition, click . You must be an **Administrator** or project **Owner** to assign quality objective definitions or thresholds to a project. You can also assign quality objectives by right-clicking a project in the **Project Explorer**.
- The **Remaining** number of findings that you need to address to reach the threshold. Click this number to open the **Review** perspective and see these findings in the **Results List**.

For a more comprehensive view, open the **Quality Objectives** dashboard. In the **Summary** section, click in the card on the left to pick a threshold and see the remaining open issues, including a breakdown for each category, such as code metrics or coding rules.

In this **Quality Objectives** dashboard, 7% of the findings required to achieve threshold `Exhaustive` have been addressed, include 38% of **Defects**. There are 1983 open issues, which are split between **Code Metrics** (55), **Coding Rule** (1676), and **Defects** (252).

This table shows the current progress of code quality for all quality objective thresholds. To view the **Results List** for a set of open issues, click the corresponding value in the table.



Group	SQO1	SQO2	SQO3	SQO4	SQO5	SQO6	Exhaustive
SQO progress	0%	12%	12%	29%	29%	34%	7%
Total Open Issues	55	109	109	165	165	307	1,983
Code Metrics	55	55	55	55	55	55	55
Coding Rules	–	–	–	–	–	–	1,676
Defects	–	54	54	110	110	252	252

Additionally, you can view aggregated SQO statistics by selecting an entire project folder in the **Project Explorer** pane. Note these differences between viewing SQO information at the folder level and the project level.

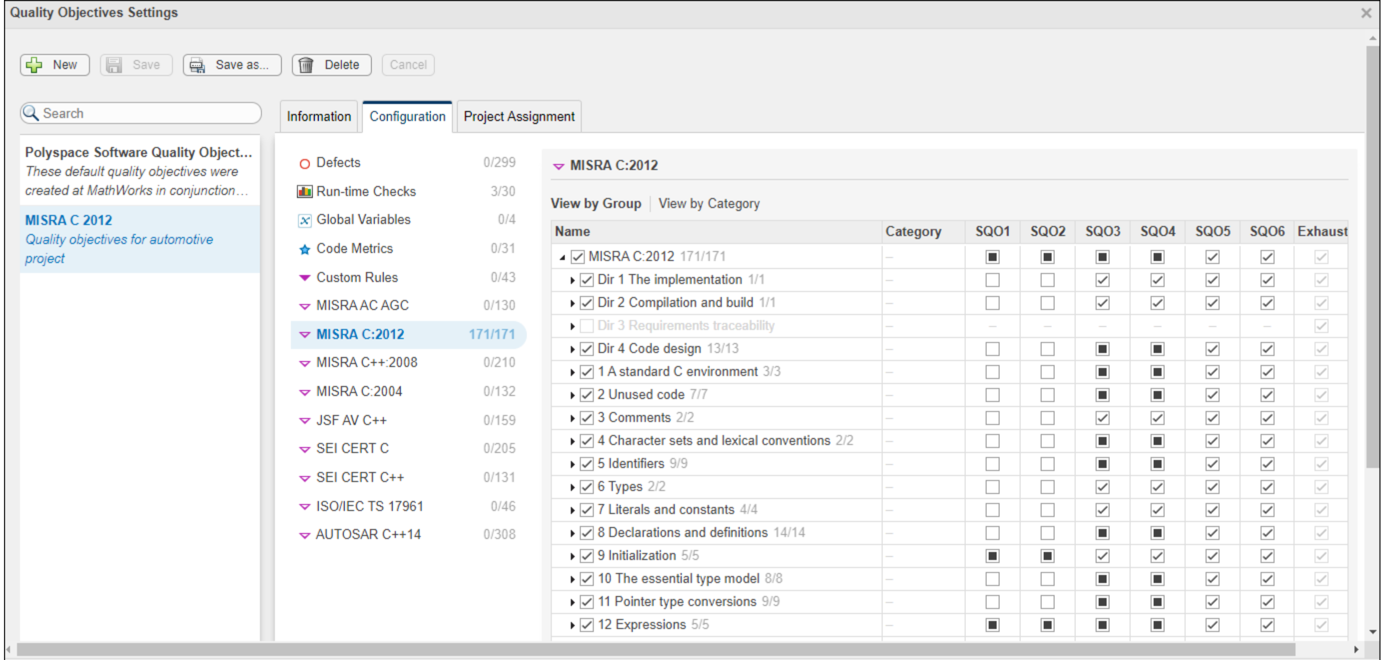
- For folders, the **Progress** and **Details** sections do not contain links to filtered results in the tables.
- You cannot assign quality objective levels to all projects in the folder at once.
- The **Quality Objectives** card for folders does not show **Pass**, **In progress**, or **Incomplete** labels.

Polyspace Access aggregates SQO statistics even if the quality objectives configuration levels are not homogeneous. While individual projects might contain separate definitions of each SQO level, Polyspace Access does not separate the statistics by level details when aggregating the statistics. All SQO1 level projects are aggregated together, as are SQO2 and so forth.

Customize Software Quality Objectives

To customize the thresholds that you use as pass or fail criteria to track the quality of your code, create or edit quality objective definitions and apply these definitions to specific projects. For instance, you might have a project where you want to check the quality of your code against only the MISRA C:2012 coding standard.

To open the quality objectives settings, click **Quality Objectives Settings** on the **Quality Objectives** dashboard.



Create Quality Objectives Definition

To create a quality objectives definition, click **New** and enter a name for the new definition. You can optionally provide a description for the quality objectives definition and for the different SQO levels of that definition. Go to the **Information** tab to view or make additional edits to the descriptions.

After you assign this definition to a project, the name of the definition is displayed on the card in the summary section of the **Quality Objectives** dashboard for that project.

Edit Quality Objectives Definition

You can edit quality objective definitions only if you have a Polyspace Access role of **Administrator** or **Owner**. To set user roles, see "Manage Project Permissions".

This table lists the different Polyspace Access roles and their corresponding write permissions for the quality object definitions.

Project Role	Write Permission
Administrator	You can edit any quality objective definition.
Owner	You can edit the quality objective definitions that you created.
Contributor	you have a read-only view of the quality objective settings and cannot make edits.

You cannot edit the default **Polyspace Software Quality Objectives**, no matter your role.

To edit the thresholds selection for a quality objectives definition:

1. Select the definition in the left pane of the **Configuration** tab.
2. Click a findings family, for instance, MISRA C:2004.

   To choose individual results, select or expand the nodes. For each family of results, you can view the nodes by group, or by category when available.

When you select nodes in the leftmost part of the table:

- ☑ indicates that all entries under the node are enabled.
- ■ indicates that some entries under the node are not enabled.

For the quality objective thresholds under the SQO columns:

- ☑ indicates that all the entries that are enabled under the node on that row apply to this threshold.
- ■ indicates that some of the entries that are enabled under the node on that row do not apply to this threshold.

	Category	SQO1	SQO2	SQO3	SQO4	SQO5	SQO6	Exhaus
■ MISRA C:2004 52/131	–	■	■	■	■	☑	☑	☑
☐ 1 Environment 0/1	–	–	–	–	–	–	–	☑
■ 2 Language extensions 2/3	–	☐	☐	☐	■	☑	☑	☑
☐ 2.1 Assembly language shall be encaps...	Required	–	–	–	–	–	–	☑
☑ 2.2 source code shall only use /* ... */ st...	Required	☐	☐	☐	☐	☑	☑	☑
☑ 2.3 The character sequence /* shall not ...	Required	☐	☐	☐	☑	☑	☑	☑

For example, in the previous figure, looking at the **Language extensions** node:

- The leftmost part of the table is marked as ■ because rule 2.1 is not enabled.
- The SQO5 and SQO6 entries along the row of the node are marked as ☑ because all the rules that are enabled under the node apply to these SQO thresholds.

- The SQO4 entry along the row of the node is marked as ▣ because rule 2.2 is enabled but does not apply to this SQO threshold.

These results are customizable by specifying numerical inputs:

- **Run-time Checks** — Customize the percentage of findings that you must address or justify for each threshold. Enter a value between 0 and 100. To disable the selection, leave the entry blank.
- **Code Metrics** — Customize the value of the different metrics for each threshold. To disable the selection, leave the entry blank.

When you make a selection for a threshold, all higher thresholds inherit that selection. For instance, if you select a coding rule for SQO3, the rule is also selected for SQO4, SQO5, and SQO6. By default, when you first select a node or an individual result, the selection applies only to SQO6.

To save your changes, click **Save**. You can also edit a quality objective definition by creating a copy of the definition using the **Save as** button and making edits to that copy. You might want to create a copy if:

- You do not have write permissions for a quality objective definition.
- You want to edit a quality objective definition but apply the changes to only your project.
- You want to use an existing definition as a template.

If you make changes to a quality objectives definition that applies to multiple projects, Polyspace Access displays a warning with a link to the **Project Assignment** tab on the **Quality Objectives Settings** window. Open the tab to determine which projects are affected by your changes and inform users that have access to those projects of your changes.

See Also

More About

- "Evaluate Polyspace Code Prover Results Against Software Quality Objectives" on page 31-2
- "Code Metrics"

Results List in Polyspace Access Web Interface

This topic focuses on the Polyspace Access web interface. To learn about the equivalent pane in the Polyspace desktop user interface, see *"Results List in Polyspace Desktop User Interface" on page 21-12**.*

The **Results List** pane lists all results along with their attributes.

For each result, the **Results List** pane contains the result attributes, listed in columns:

Attribute	Description
Family	Group to which the result belongs.
ID	Unique identification number of the result.
Type	Defect or coding rule violation.
Group	Category of the result, for instance: • For defects: Groups such as static memory, numerical, control flow, concurrency, etc. • For coding rule violations: Groups defined by the coding rule standard. For instance, MISRA C: 2012 defines groups related to code constructs such as functions, pointers and arrays, etc.
Check	Result name, for instance: • For defects: Defect name • For coding rule violations: Coding rule number
Information	Result sub-type when available. • For defects: Impact classification. For coding standards: required or mandatory, rule or recommendation.
Detail	Additional information about a result. The column shows the first line of the **Result Details** pane. For an example of how to use this column, see the result `MISRA C:2012 Dir 1.1`.
File	File containing the instruction where the result occurs
Function	Function containing the instruction where the result occurs. If the function is a method of a class, it appears in the format `class_name::function_name`.

Attribute	Description
Status	Review status you have assigned to the result. The possible statuses are: • `Unreviewed` (default status) • `To investigate` • `To fix` • `Justified` • `No action planned` • `Not a defect` • `Other` See also "Add Custom Status in Polyspace Access Project" on page 27-3.
Severity	Level of severity you have assigned to the result. The possible levels are: • `Unset` • `High` • `Medium` • `Low`
Assigned to	User name of reviewer assigned to this result.
Ticket Key	When you create a bug tracking tool (BTT) ticket for a result, this field contains the ticket ID. Click the ticket ID in the **Results Details** to open the ticket in the BTT interface.
Comments	Comments you have entered about the result
Folder	Path to the folder that contains the source file with the result

To show or hide any of the columns, click the icon in the upper-right of the **Results List** pane, then select or clear the title of the column that you want to show or hide.

Using this pane, you can:

- Navigate through the results.
- Organize your result review using filters in the toolstrip or in the context menu. For more information, see "Filter and Sort Results in Polyspace Access Web Interface" on page 28-8.
- Right-click a result to get the URL of the result. When you open this URL in a web browser you get see the **Results List** pane filtered to that one result.

If the **Results List** exceeds 10000 findings, Polyspace Access truncates the list and displays this icon in the filters bar. To show all findings, see the contextual help of the icon.

The 10000 findings limit is preset and cannot be changed.

Source Code in Polyspace Access Web Interface

This topic focuses on the Polyspace Access web interface. To learn about the equivalent pane in the Polyspace desktop user interface, see *“Source Code in Polyspace Desktop User Interface” on page 21-15**.*

The **Source Code** pane shows the source code with the defects colored in red.

```
Source Code
numerical.c ×   programming.c ×
306   *  INVALID USE OF INTEGER STANDARD LIBRARY
307   *=====================================================================
308  div_t bug_intstdlib(int num, int denom) {
309      div_t test;
310
311      if (denom == 0)
312          test = div(num, denom);            /* Defect: Generates a divis
313 M    else if ((num == INT_MIN) && (denom == -1))
314          test = div(num, denom); /* Defect: Generates an overflow on fir
315      else
316          test = div(num, denom);
317
318      return test;
319  }
320
321  div_t corrected_intstdlib(int num, int denom) {
```

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 Code** 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, you can use the following options to examine and navigate through your code:

- **Search For All References** — List all references in the **Code Search** pane. The software supports this feature for global and local variables, functions, types, and classes.
- **Go To Definition** — Go to the line of code that contains the definition of `i`. The software supports this feature for global and local variables, functions, types, and classes. If a definition is not available to Polyspace, selecting the option takes you to the declaration.
- **Select Results** –– Show more information about the selected result in the **Results Details** pane and pin the result in the **Source Code** pane.

  After you navigate away from the current result, use the icon on the **Result Details** pane to come back.
- **Go To Line** — Open the Go to line dialog box. If you specify a line number and click **Enter**, the software displays the specified line of code.

To search for instances of your selection in the **Current Source File** or in **All Source Files**, double-click your selection before you right-click.

Expand Macros

You can view the contents of source code macros in the source code view. A code information bar displays icons that identify source code lines with macros.

When you click this icon, the software displays the contents of macros on the next line.

```
Source Code
numerical.c ×   programming.c ×
306   *  INVALID USE OF INTEGER STANDARD LIBRARY
307   *=====================================================================
308  div_t bug_intstdlib(int num, int denom) {
309      div_t test;
310
311      if (denom == 0)
312          test = div(num, denom);          /* Defect: Generates a divis
313 M    else if ((num == INT_MIN) && (denom == -1))
         else if ((num == (-0x7fffffffL - 1)) && (denom == -1))
314          test = div(num, denom); /* Defect: Generates an overflow on fir
315      else
316          test = div(num, denom);
317
318      return test;
319  }
320
321  div_t corrected_intstdlib(int num, int denom) {
```

To display the normal source code again, click the icon again.

Note

1 The **Result Details** pane also allows you to view the contents of a macro if the check you select lies within a macro.

2 You cannot expand OSEK API macros in the **Source Code** pane.

View Code Block

On the **Source Code** pane, to highlight a block of code, click either its opening or closing brace. If the brace itself is highlighted, click the brace twice.

Source Code

numerical.c × programming.c ×

```
306   *  INVALID USE OF INTEGER STANDARD LIBRARY
307   *=====================================================================
308  div_t bug_intstdlib(int num, int denom) {
309      div_t test;
310
311      if (denom == 0)
312          test = div(num, denom);            /* Defect: Generates a divis
313      else if ((num == INT_MIN) && (denom == -1))
314          test = div(num, denom); /* Defect: Generates an overflow on fir
315      else
316          test = div(num, denom);
317
318      return test;
319  }
320
321  div_t corrected_intstdlib(int num, int denom) {
```

Navigate from Code to Model

If you run Polyspace on generated code in Simulink and upload the results to Polyspace Access, you can navigate from the source code in Polyspace Access to blocks in the model.

On the **Source Code** pane in the Polyspace Access web interface, links in code comments show blocks that generate the subsequent lines of code. To see the block in the model:

- Right-click a link and select **Copy MATLAB Command to Highlight Block**.

This action copies the MATLAB command required to highlight the block. The command uses the `Simulink.ID.hilite` function.

- In MATLAB editor, paste and run the copied command with the model open.

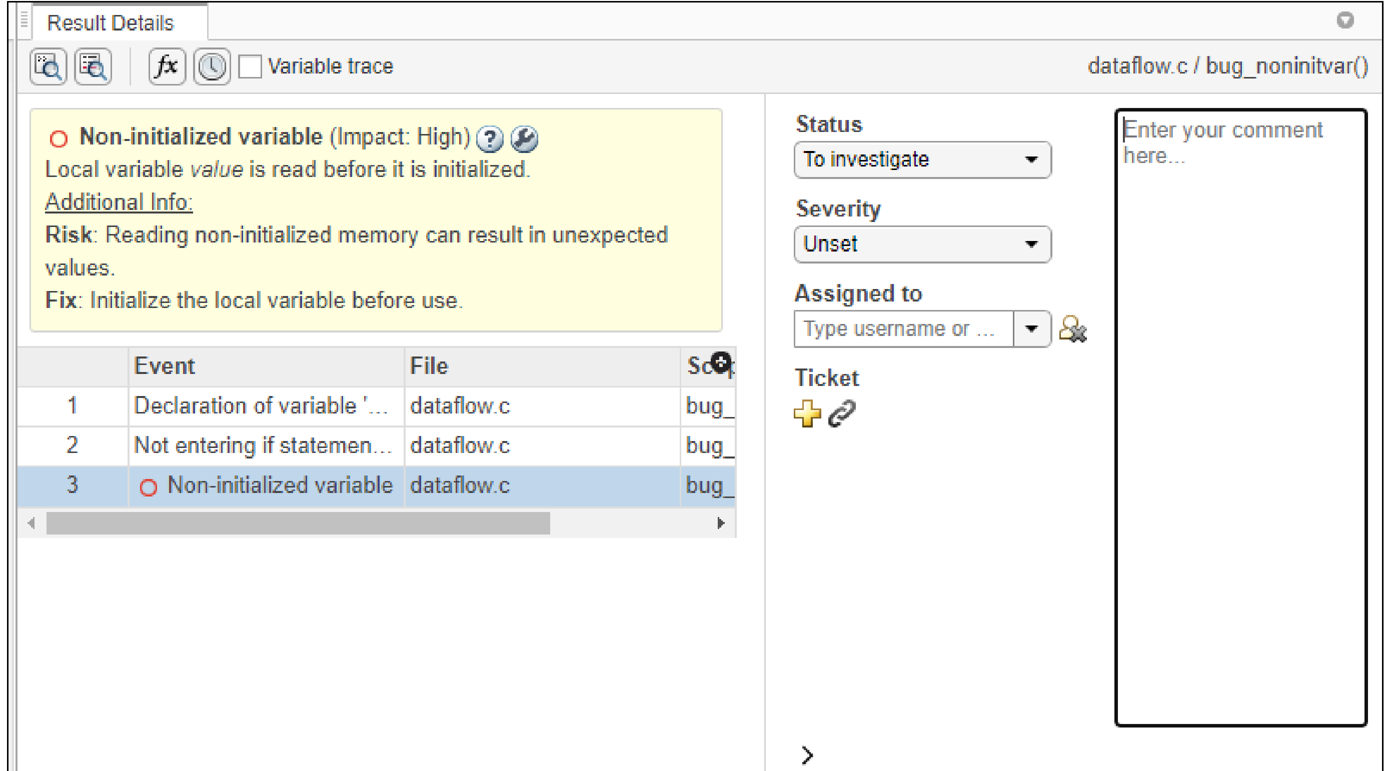
Result Details in Polyspace Access Web Interface

This topic focuses on the Polyspace Access web interface. To learn about the equivalent pane in the Polyspace desktop user interface, see "Result Details in Polyspace Desktop User Interface" on page 21-21.

The **Result Details** pane contains comprehensive information about a specific defect. To see this information, on the **Results List** pane, select the defect.

- The top right corner shows the file and function containing the defect, in the format `file_name/function_name`.
- The yellow box contains the name of the defect with an explanation of why the defect occurs.

  The button allows you to access documentation for the defect. When available, click the icon to see fix suggestions for the defect.



On this pane, you can also:

- Assign a **Severity** and **Status** to each check, and enter comments to describe the results of your review.
- Assign a reviewer to the result. A reviewer can filter the **Results List** to only show results that are assigned to him or her.
- Create a ticket in a bug tracking tool such as JIRA. Once you create the ticket the **Results Details** for this defect shows a clickable link to the ticket you created.
- View the event traceback.

The **Event** column lists the sequence of code instructions causing the defect. The **Scope** column lists the function containing the instructions. If the instructions are not in a function, the column lists the file containing the instructions.

The **Variable trace** check box allows you to see an additional set of instructions that are related to the defect.

- Click the icon to open the “Call Hierarchy in Polyspace Access Web Interface” on page 26-26.
- Click the icon to open the “Review History in Polyspace Access Web Interface” on page 26-36.
- Click the icon to open the:
  - **Error Call Graph** if the selected finding is a **Run-time Check**.

    The pane displays the call sequence that leads to the detected finding. Click a node on the graph to navigate back to the source code.
  - **Variable Access Graph** if the selected finding is a **Global variable**.

    The pane displays a graphical representation of the access operations on global variables. Click a node on the graph to navigate back to the source code at the location of calling and called functions.

See Also

More About

- “Address Results in Polyspace Access Through Bug Fixes or Justifications”
- “Review History in Polyspace Access Web Interface”

Call Hierarchy in Polyspace Access Web Interface

This topic focuses on the Polyspace Access web interface. To learn about the equivalent pane in the Polyspace desktop user interface, see "Call Hierarchy in Polyspace Desktop User Interface" on page 21-24.

The **Call Hierarchy** pane displays the call tree of functions in the source code.

For each function `foo`, the **Call Hierarchy** pane lists the functions and tasks that call `foo` (callers) and those called by `foo` (callees). The callers are indicated by . The callees are indicated by . The **Call Hierarchy** pane lists direct function calls and indirect calls through function pointers.

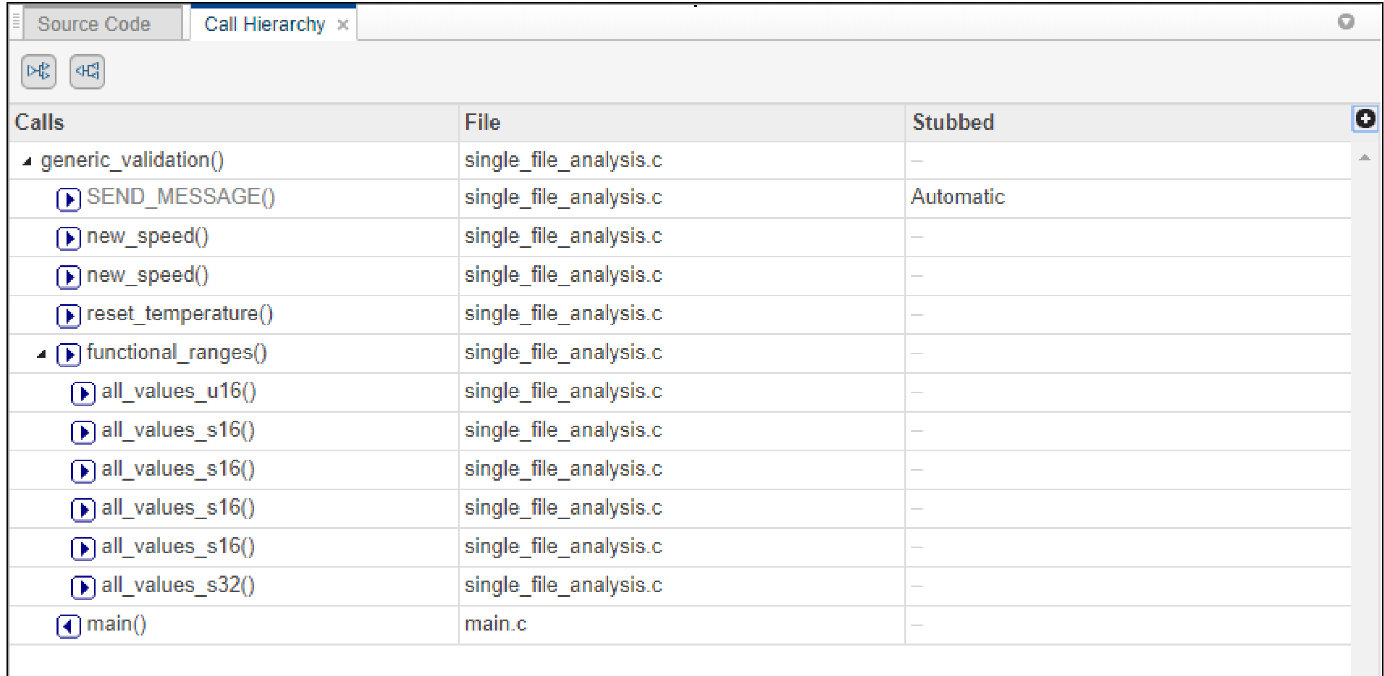
Note For Polyspace Access findings, you might not see all callers or callees of a function, especially for calls through function pointers and dead code.

For instance, Polyspace Access does not display the functions registered with `at_exit()` and `at_quick_exit()`, and called by `exit()` and `quick_exit()` respectively.

You open the **Call Hierarchy** pane by using the icon in your **Results Details** pane, or by going to **Window > Call Hierarchy**.

To update the pane, click a defect on the **Results List** or CTRL-click a result in the **Source Code** pane. You see the function containing the defect with its callers and callees.

In this example, the **Call Hierarchy** pane displays the function `generic_validation`, and with its callers and callees.

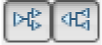


Tip To navigate to the call location in the source code, select a caller or callee name

In the **Call Hierarchy** pane, you can perform these actions:

- **Show/Hide Callers and Callees**

  Customize the view to display callers only or callees only. Show or hide callers and callees by clicking this button

- **Navigate Call Hierarchy**

  You can navigate the call hierarchy in your source code. For a function, double-click a caller or callee name to navigate to the caller or callee definition in the source code.

- **Determine If Function Is Stubbed**

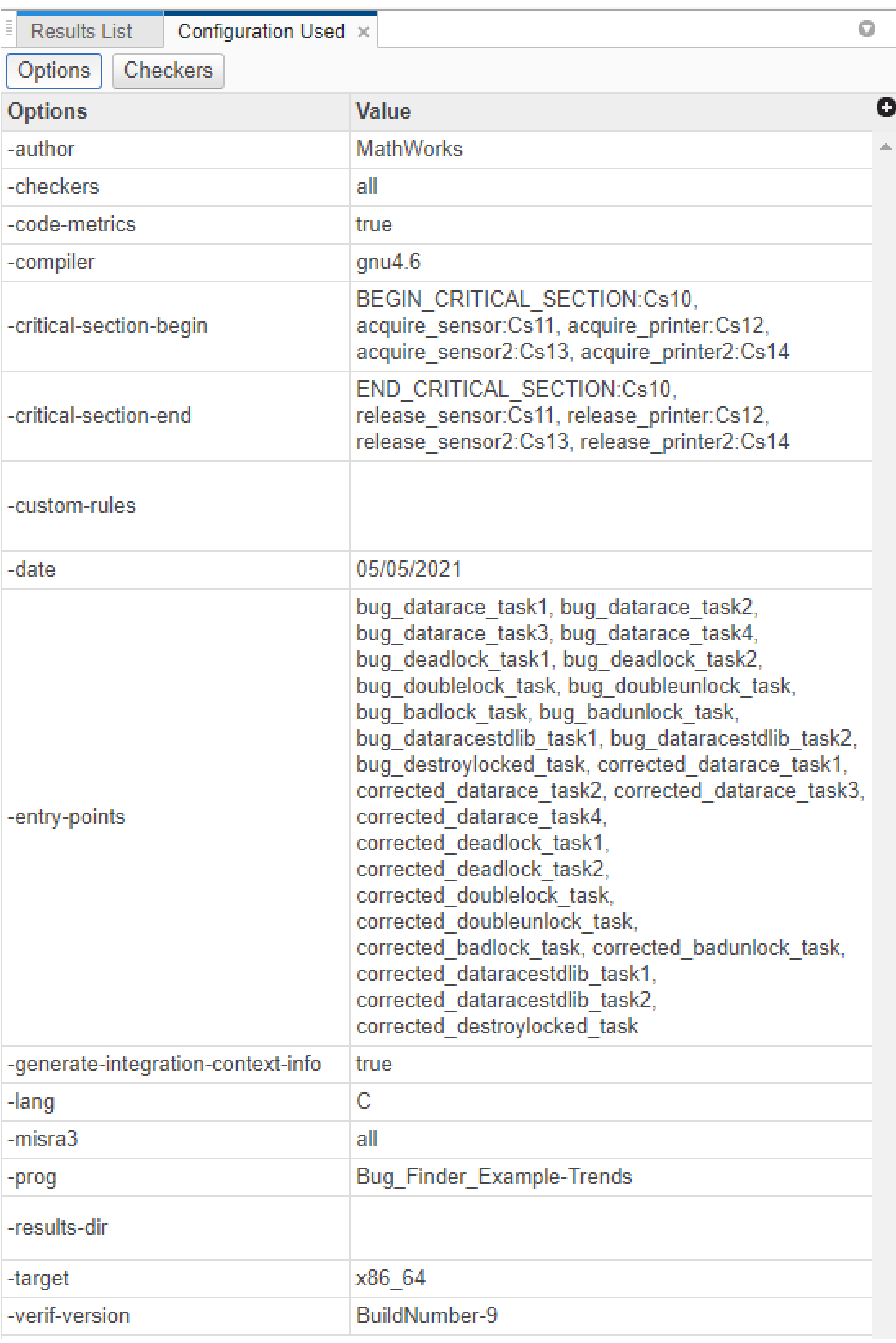
  From the **Stubbed** column, you can determine if a function is stubbed. The entries in the column show why a function was stubbed.

  - **Automatic**: Polyspace cannot find the function definition. For instance, you did not provide the file containing the definition.
  - **Std library**: The function is a standard library function. You do not provide the function definition explicitly in your Polyspace project.
  - **Mapped to std library**: You map the function to a standard library function by using the option `-code-behavior-specifications`.

Configuration Settings in Polyspace Access Web Interface

The **Configuration Used** pane displays the options and checkers that were enabled to generate the results in the currently selected project. The **Options** tab shows user-specified options and options that are enabled by default.

You open the **Configuration Used** pane by going to **Window** > **Configuration Used**.



Options	Value
-author	MathWorks
-checkers	all
-code-metrics	true
-compiler	gnu4.6
-critical-section-begin	BEGIN_CRITICAL_SECTION:Cs10, acquire_sensor:Cs11, acquire_printer:Cs12, acquire_sensor2:Cs13, acquire_printer2:Cs14
-critical-section-end	END_CRITICAL_SECTION:Cs10, release_sensor:Cs11, release_printer:Cs12, release_sensor2:Cs13, release_printer2:Cs14
-custom-rules	
-date	05/05/2021
-entry-points	bug_datarace_task1, bug_datarace_task2, bug_datarace_task3, bug_datarace_task4, bug_deadlock_task1, bug_deadlock_task2, bug_doublelock_task, bug_doubleunlock_task, bug_badlock_task, bug_badunlock_task, bug_dataracestdlib_task1, bug_dataracestdlib_task2, bug_destroylocked_task, corrected_datarace_task1, corrected_datarace_task2, corrected_datarace_task3, corrected_datarace_task4, corrected_deadlock_task1, corrected_deadlock_task2, corrected_doublelock_task, corrected_doubleunlock_task, corrected_badlock_task, corrected_badunlock_task, corrected_dataracestdlib_task1, corrected_dataracestdlib_task2, corrected_destroylocked_task
-generate-integration-context-info	true
-lang	C
-misra3	all
-prog	Bug_Finder_Example-Trends
-results-dir	
-target	x86_64
-verif-version	BuildNumber-9

Click **Checkers** to see which checkers are enabled for:

- “Coding Standards”, for instance MISRA C: 2012.
- “Custom Coding Rules”.

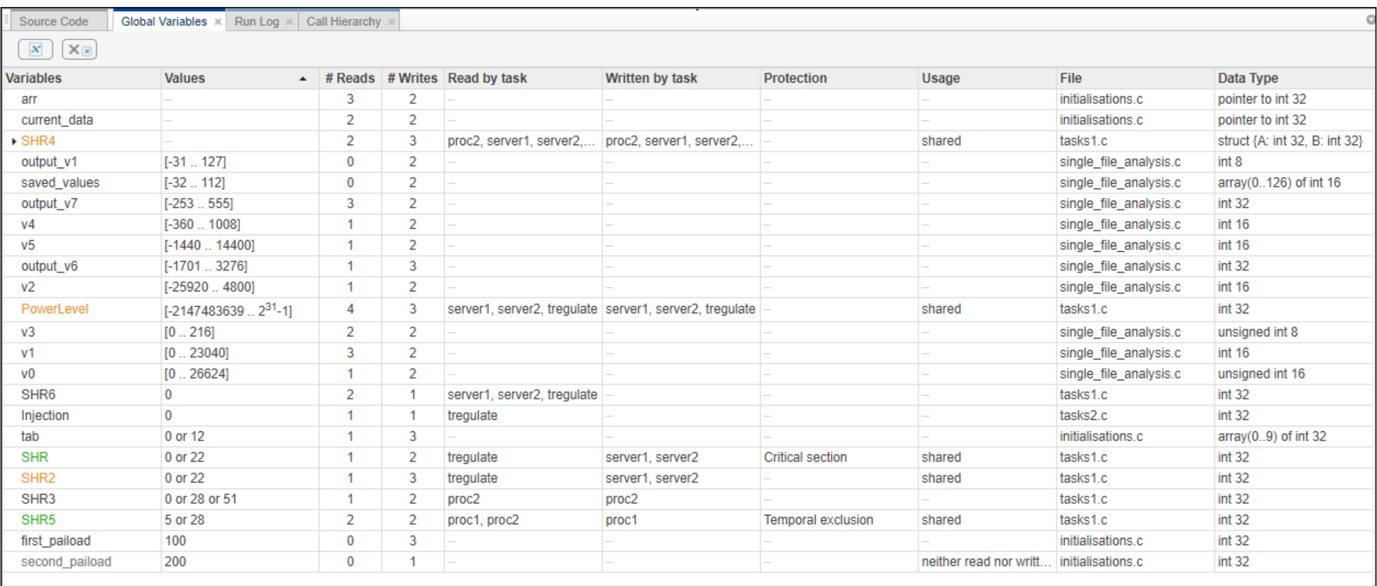
Checkers is not available for a Code Prover project if no coding standard or custom coding rules are enabled.

Global Variables in Polyspace Access Web Interface

This topic focuses on the Polyspace Access web interface. To learn about the equivalent pane in the Polyspace desktop user interface, see "Variable Access in Polyspace Desktop User Interface" on page 21-27.

The **Global Variables Usage** pane displays global variables (and local static variables). For each global variable, the pane lists all functions and tasks performing read/write access on the variables, along with their attributes, such as values, read/write accesses and shared usage.

You open the **Global Variables Usage** pane by using the icon in your **Results Details** pane, or by going to **Window > Global Variables Usage**.



For each variable and each read/write access, the **Global Variables Usage** pane contains the relevant attributes. For the variables, the various attributes are listed in this table.

Attribute	Description
Variables	Name of Variable
Values	Value (or range of values) of variable This column is empty for pointer variables.
# Reads	Number of times the variable is read
# Writes	Number of times the variable is written
Read by task	Name of tasks reading variable
Written by task	Name of tasks writing on variable

Attribute	Description
Protection	Whether shared variable is protected from concurrent access (Filled only when **Usage** column has entry, **Shared**) The possible entries in this column are: • **Critical Section**: If variable is accessed in critical section of code • **Temporal Exclusion**: If variable is accessed in mutually exclusive tasks For more details on these entries, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.
Usage	`Shared`, if variable is shared between tasks; otherwise, blank
File	Source file containing variable declaration
Data Type	Data type of variable (C/C++ data types or structures/classes)

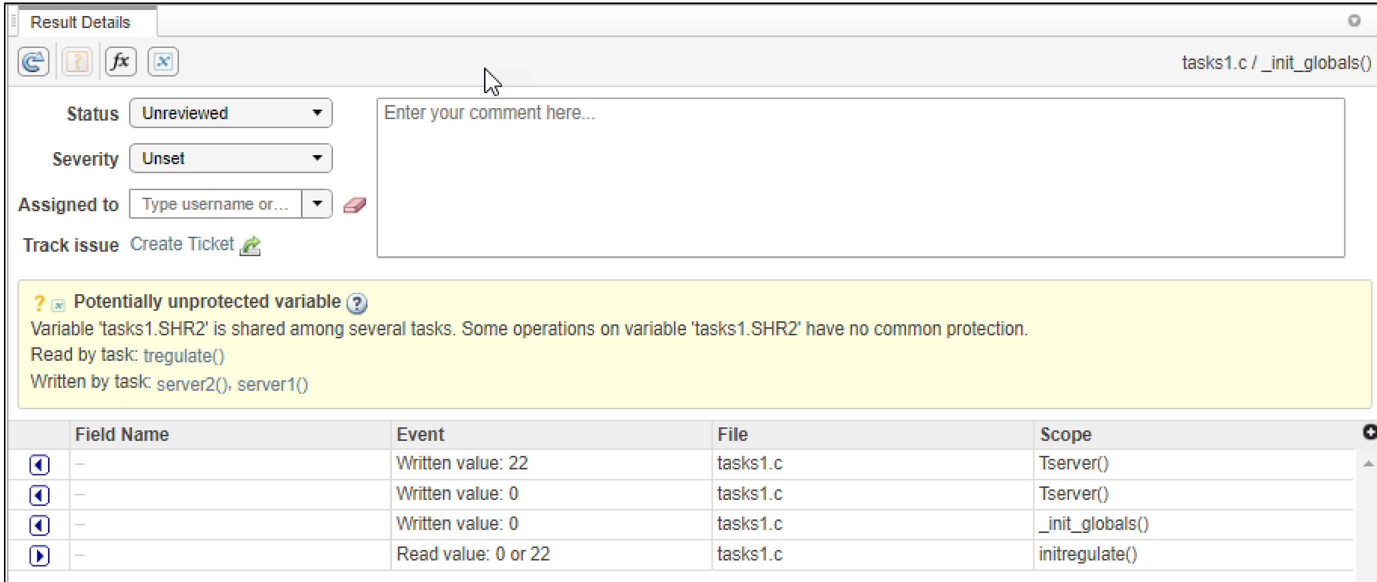
Double-click a variable name to view read/write access operations on the variable in the **Results Details** pane. The arrowhead symbols and in the **Results Details** pane indicate functions performing read and write access respectively on the global variable. For further information on tasks, see analysis option `Tasks (-entry points)` in the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

For access operations on the variables, the various attributes described in the **Global Variables Usage** pane are listed in this table.

Attribute	Description
Values	Value or range of values of variable in the function or task performing read/write access This column is empty for pointer variables.
Written by task	*Only for tasks*: Name of task performing write access on variable
Read by task	*Only for tasks*: Name of task performing read access on variable
File	Source file containing access operation on variable

The **Results Details** pane also lists the **Scope** of the access operation on the variable.

For example, consider the global variable, `SHR2`:



The function, `Tserver`, in the file, `tasks1.c`, performs two write operations on SHR2. This is indicated in the **Results Details** pane by the two instances of `Tserver()` in the table, marked by . Likewise, the read access by task `initregulate` is also listed in the table and marked by .

The color scheme for variables in the **Global Variables Usage** pane is:

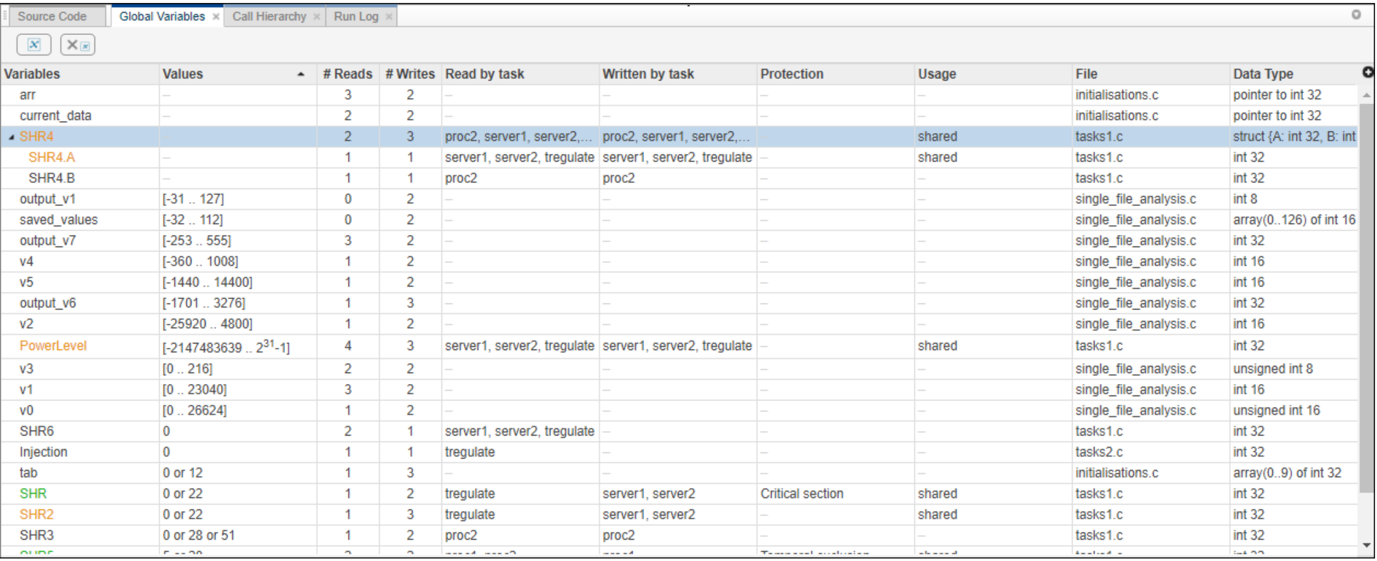
- Black: global variable.
- Orange: global variable, shared between tasks with no protection against concurrent access.
- Green: global variable, shared between tasks and protected against concurrent access.
- Gray: global variable, declared but not used in reachable code.

If a task performs certain operations on a global variable, but the operations are in unreachable code, the tasks are colored gray.

The information about global variables and read/write access operations obtained from the **Global Variables Usage** pane is called the data dictionary.

You can also perform the following actions from the **Global Variables Usage** pane.

- **View Structured Variables**



Variables	Values	# Reads	# Writes	Read by task	Written by task	Protection	Usage	File	Data Type
arr	–	3	2	–	–	–	–	initialisations.c	pointer to int 32
current_data	–	2	2	–	–	–	–	initialisations.c	pointer to int 32
SHR4	–	2	3	proc2, server1, server2,...	proc2, server1, server2,...	–	shared	tasks1.c	struct {A: int 32, B: int
SHR4.A	–	1	1	server1, server2, tregulate	server1, server2, tregulate	–	shared	tasks1.c	int 32
SHR4.B	–	1	1	proc2	proc2	–	–	tasks1.c	int 32
output_v1	[-31 .. 127]	0	2	–	–	–	–	single_file_analysis.c	int 8
saved_values	[-32 .. 112]	0	2	–	–	–	–	single_file_analysis.c	array(0..126) of int 16
output_v7	[-253 .. 555]	3	2	–	–	–	–	single_file_analysis.c	int 32
v4	[-360 .. 1008]	1	2	–	–	–	–	single_file_analysis.c	int 16
v5	[-1440 .. 14400]	1	2	–	–	–	–	single_file_analysis.c	int 16
output_v6	[-1701 .. 3276]	1	3	–	–	–	–	single_file_analysis.c	int 32
v2	[-25920 .. 4800]	1	2	–	–	–	–	single_file_analysis.c	int 16
PowerLevel	[-2147483639 .. 2^{31}-1]	4	3	server1, server2, tregulate	server1, server2, tregulate	–	shared	tasks1.c	int 32
v3	[0 .. 216]	2	2	–	–	–	–	single_file_analysis.c	unsigned int 8
v1	[0 .. 23040]	3	2	–	–	–	–	single_file_analysis.c	int 16
v0	[0 .. 26624]	1	2	–	–	–	–	single_file_analysis.c	unsigned int 16
SHR6	0	2	1	server1, server2, tregulate	–	–	–	tasks1.c	int 32
Injection	0	1	1	tregulate	–	–	–	tasks2.c	int 32
tab	0 or 12	1	3	–	–	–	–	initialisations.c	array(0..9) of int 32
SHR	0 or 22	1	2	tregulate	server1, server2	Critical section	shared	tasks1.c	int 32
SHR2	0 or 22	1	3	tregulate	server1, server2	–	shared	tasks1.c	int 32
SHR3	0 or 28 or 51	1	2	proc2	proc2	–	–	tasks1.c	int 32

For structured variables, double click the variable in the **Global Variables Usage** pane to view the individual fields. For example, for the structure, `SHR4`, the pane displays the fields, `SHR4.A` and `SHR4.B`, and the functions performing read/write access on them.

- **Show/Hide Callers and Callees**

Customize the **Global Variables Usage** pane to show only the shared variables. On the **Global Variables Usage** pane toolbar, click the Non-Shared Variables button to show or hide non-shared variables.

- **Hide Access in Unreachable Code**

Hide read/write access occurring in unreachable code by clicking the filter button .

- **Limitations**

You cannot see an addressing operation on a global variable or object (in C++) as a read/write operation in the **Global Variables Usage** pane. For example, consider the following C++ code:

```
class C0
{
public:
  C0() {}
  int get_flag()
  {
    volatile int rd;
    return rd;
  }
  ~C0() {}
private:
  int a;                /* Never read/written */
};

C0 c0;                  /* c0 is unreachable */

int main()
{
  if (c0.get_flag())    /* Uses address of the method */
    {
      int *ptr = take_addr_of_x();
```

```
      return 1;
    }
  else
    return 0;
}
```

You do not see the method call `c0.get_flag()` in the **Global Variables Usage** pane because the call is an addressing operation on the method belonging to the object `c0`.

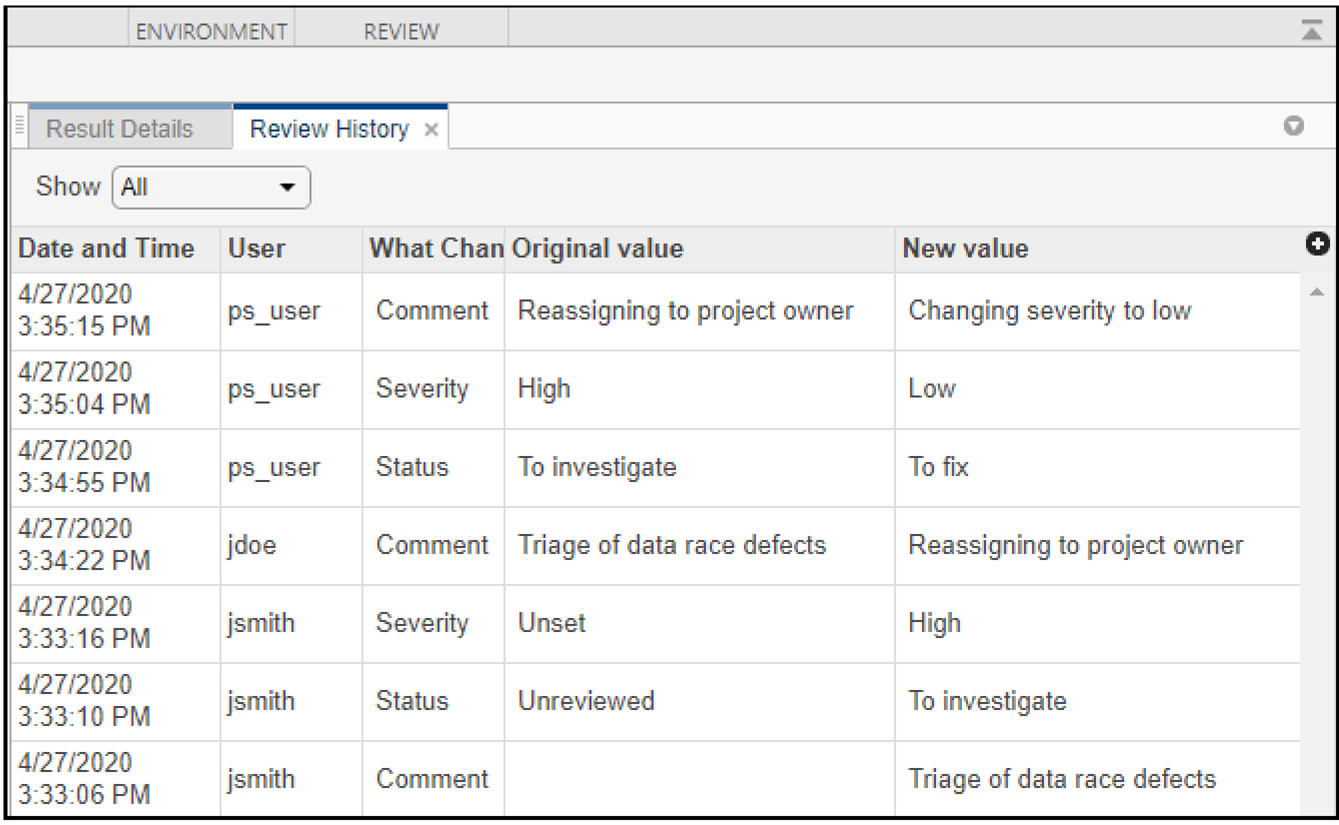
Review History in Polyspace Access Web Interface

The **Review History** pane displays changes to the **Status**, **Severity**, or **Comment** for a finding. For each change to these review fields, you see a separate row with:

- The date and time of the change.
- The user name of the user who made the change.
- The review field that changed, for instance **Severity**.
- The original value of the review field.
- The new value of the review field.

All the changes that you make to the review fields of findings in the Polyspace desktop interface are shown in a single row after you upload these findings to Polyspace Access. The **Review History** pane does not display the user name of the user who made these changes.

You open the **Review History** pane by using the icon in your **Results Details** pane, or by going to **Window** > **Review History**.



You can display changes for all the review fields, or you can filter for changes by **Status**, **Severity**, and **Comment**.

See Also

More About

- "Address Results in Polyspace Access Through Bug Fixes or Justifications"
- "Result Details in Polyspace Access Web Interface"

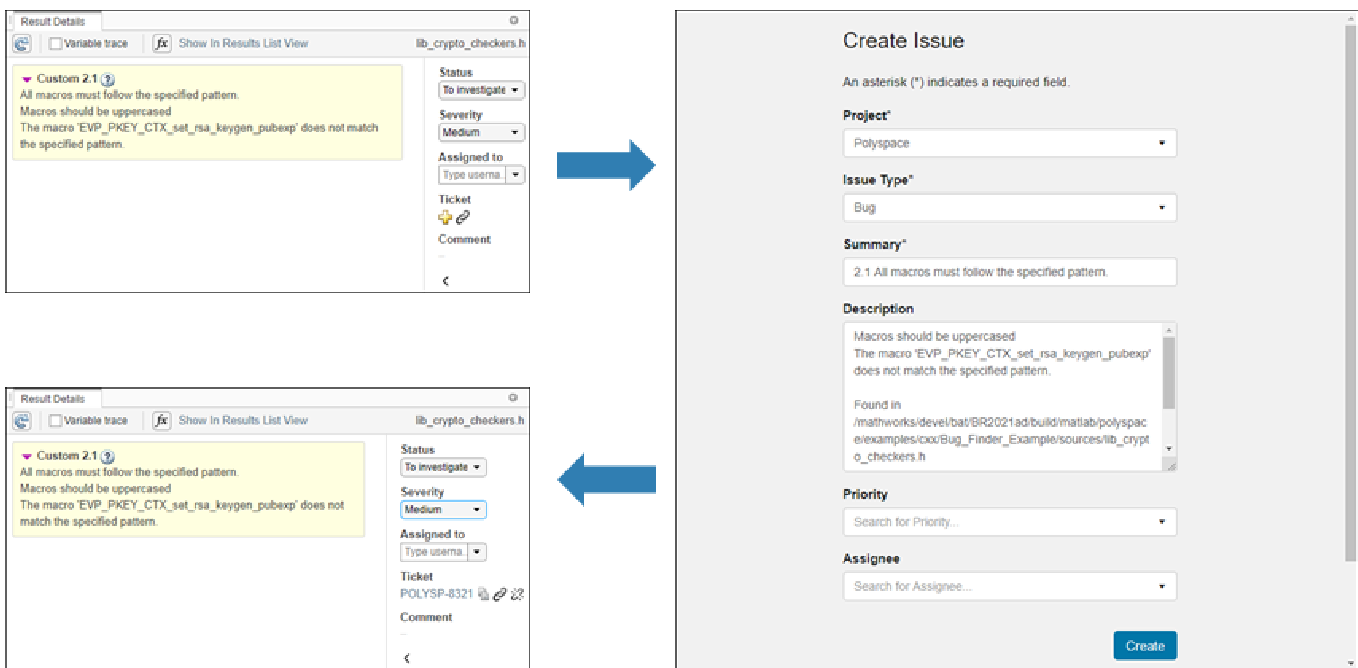
Create Bug Tracking Tool Tickets from the Polyspace Access Web Interface

If you use a bug tracking tool (BTT) such as Jira Software or Redmine as part of your software development process, you can configure Polyspace Access to create BTT tickets for Polyspace findings and add those tickets to the relevant project in your BTT software. See “Configure Issue Tracker”.

Create a Ticket

To create a BTT ticket, select one or more findings in the **Results list** and, from the **Results Details** pane, click in Polyspace Access or **Create ticket** in the Polyspace desktop interface. To select multiple findings, press **CTRL** and click the findings.

Note In the desktop interface, you can create a BTT ticket only for results that you open from Polyspace Access.



If you use Jira, you may be prompted to enter your credentials. These credentials might be different from your Polyspace Access credentials.

After you create a BTT ticket, click the link in the **Results Details** pane to open the ticket in the BTT interface and track the progress in resolving the issue. For each finding that you selected when you created the ticket, the **Description** field of the ticket includes a URL to the Polyspace Access **Results List** filtered down to that finding.

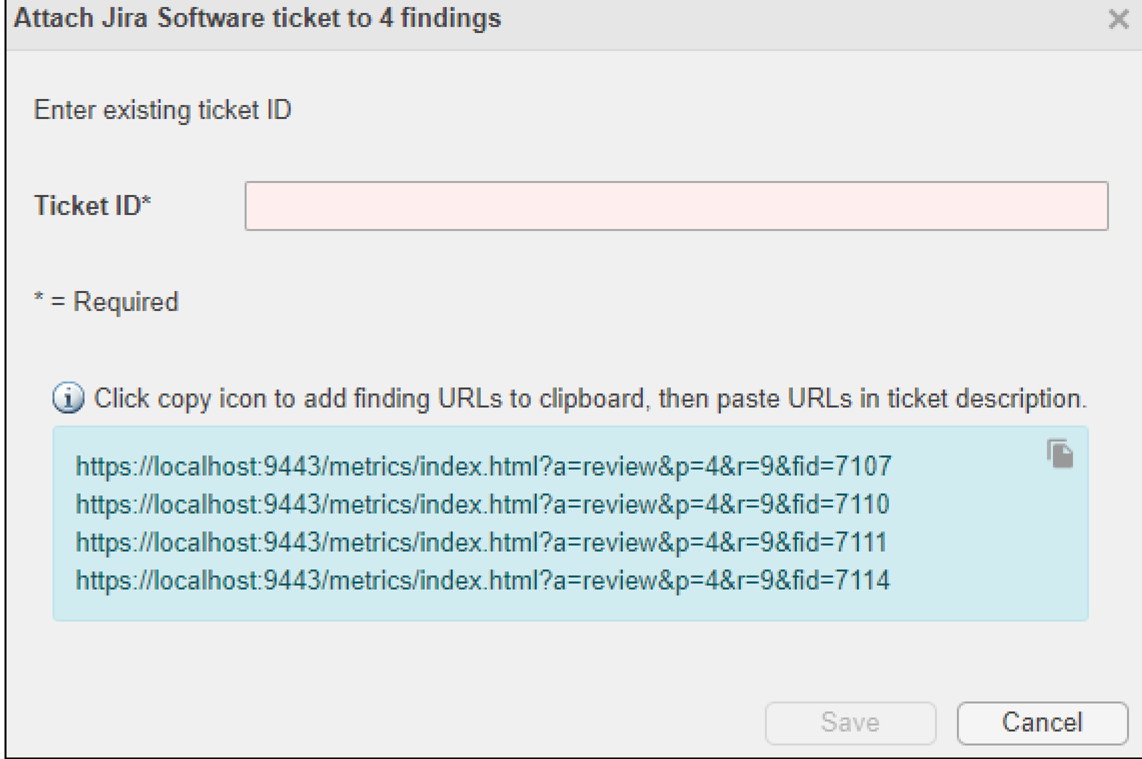
Manage Existing Tickets

Once you create a BTT ticket, you can attach the ticket to additional findings or detach the ticket from findings associated with the ticket. To attach a ticket to additional findings:

1. Select findings in the **Results List** and then click [link icon] in the **Result Details**.
2. When prompted, enter the **ticket ID** in the dialogue window.

   Click the copy icon in the **Result Details** pane of a finding already associated with the ticket to copy the **ticket ID**. The copy icon is not available when you select multiple findings with different ticket IDs. The **ticket ID** is also available in the **Ticket Key** column of the **Results List**.
3. Click the copy icon in the dialogue window to copy the findings URL, then click **Save**.
4. Click the ticket URL in the **Result Details** to open the ticket in the BTT interface and paste the findings URL you copied into the ticket description field.

You cannot attach more than one ticket to a finding. If a finding is already associated with a ticket, attaching a new ticket overwrites the existing **ticket ID**. This operation does not overwrite the ticket in your BTT. You can see all findings associated with a **ticket ID** by using the **Show only** text filter in the toolstrip.



To detach a ticket from a finding, select the finding in the **Results List**, then click [unlink icon] in the **Result Details**. The link to the ticket is removed from the **Result Details** pane. This operation does not remove the ticket in your BTT.

Note You cannot manage existing BTT tickets in the Polyspace desktop interface.

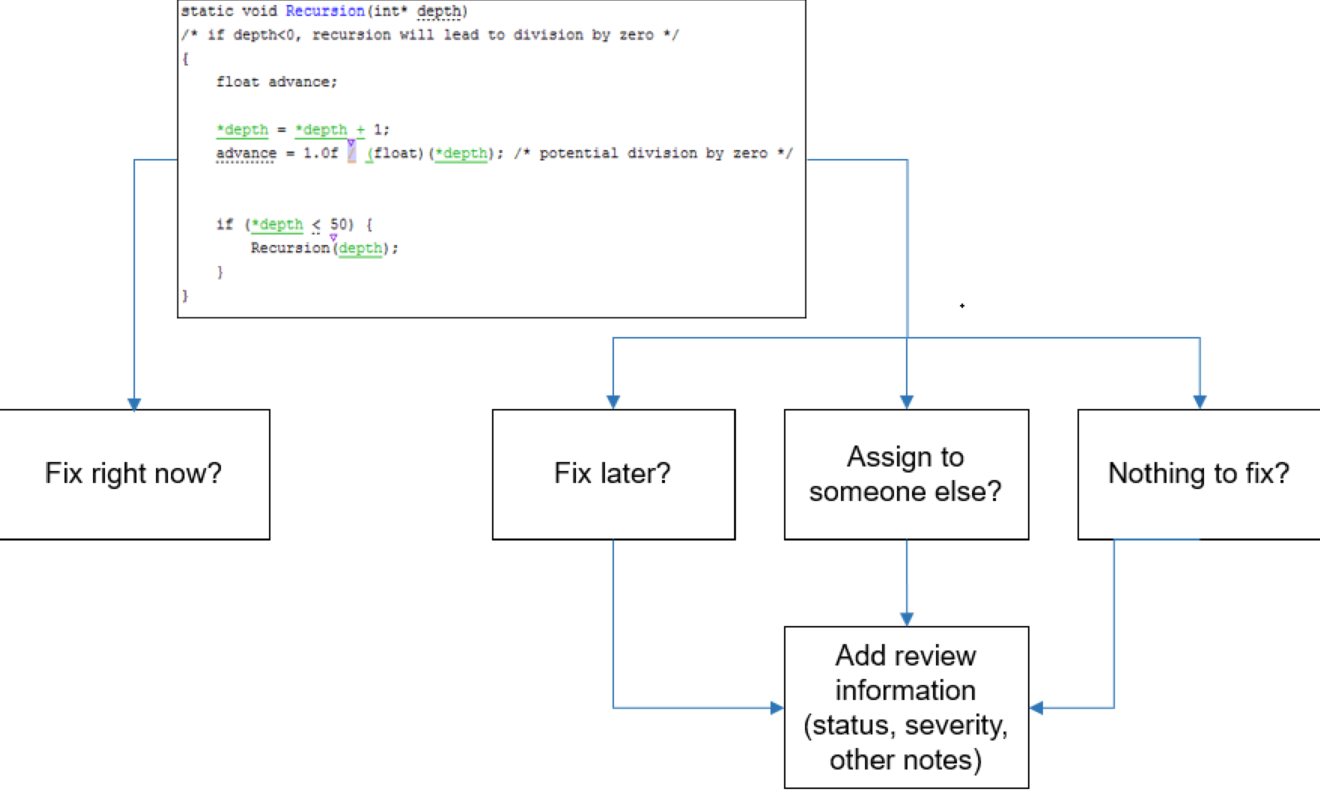
Fix or Comment Polyspace Results on Web Browser

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2
- “Import Review Information from Existing Polyspace Access Projects” on page 27-5

Address Results in Polyspace Access Through Bug Fixes or Justifications

This topic describes how to add review information to Polyspace results in the Polyspace Access web interface. For a similar workflow in the user interface of the Polyspace desktop products, see “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2.

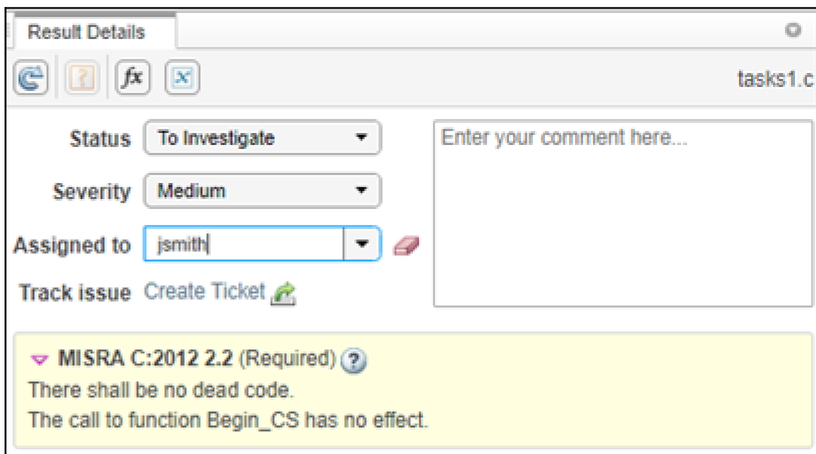
Once you understand the root cause of a Polyspace finding, you can fix your code. Otherwise, add review information to your Polyspace results to fix the code later or to justify the result. You can use the information to keep track of your review progress.



If you add review information to your results file, they carry over to the results of the next analysis on the same project. If you add the same information as comments to your code (annotate), they carry over to any subsequent analysis of the code, whether in the same project or not.

Add Review Information in Result Details pane

Set the **Status** and **Severity**, and optionally enter notes with more explanations in the **Result Details** pane. The status indicates your response to the Polyspace result.



If you do not plan to fix your code in response to a result, assign one of the following statuses:

- `Justified`
- `No Action Planned`
- `Not a Defect`

These statuses indicate that you have given due consideration and justified that result (retained the code despite the result). Note that subsequent analyses continue to show justified results as before. For instance, a Code Prover result that was previously orange does not turn green after justification. However, during review, you can filter out justified results in one click and focus only on results that are not justified. See “Filter and Sort Results in Polyspace Access Web Interface” on page 28-8.

Add Custom Status in Polyspace Access Project

If your company uses custom review statuses for results, you add those statuses in Polyspace Access. To add a custom **Status**:

1. Open the results in a Polyspace desktop interface. See “Open Polyspace Access Results in a Desktop Interface” on page 29-2.
2. In the Polyspace desktop interface, create a custom status. See “Create Custom Review Status” on page 2-27.

After you create the status, it is available from the **Status** dropdown in the **Results Details** pane. You can assign that custom status from Polyspace Access or from the Polyspace desktop interface.

Once you assign a custom status to a finding, you can apply that status to other findings in the same project. You cannot assign the custom status to findings in other projects.

Track Review Progress

To facilitate your review workflow in Polyspace Access, the findings are classified as:

- **To Do** — Findings with a status of `Unreviewed` that need to be addressed with a fix or a justification.
- **In Progress** — Findings with a status of `To fix`, `To investigate`, or `Other` that need to be addressed with a fix or a justification.
- **Done** — Findings with a status of `Justified`, `No action planned`, or `Not a defect`.

Note Green run-time checks, green shared variables, non-shared variables, and code metrics do not need to be addressed or justified. These findings do not count toward the number of findings that are **To Do**, **In Progress**, and **Done**.

In the **DASHBOARD** perspective, findings that are **To Do** or **In Progress** are considered as **Open Issues**. If a Polyspace analysis of your code finds known or acceptable defects or coding rule violations, you can remove the defects or violations from this list of **Open Issues** in subsequent analyses by assigning one of the justified statuses outlined above.

Comment or Annotate in Code

You can also add specific code comments or annotations in a code editor in response to Polyspace results. If you enter code comments or annotations in a specific syntax, on the next analysis of the code, the software can read them and populate the **Severity**, **Status**, and **Comment** fields in the result details.

For the annotation syntax, see "Annotate Code and Hide Known or Acceptable Results" on page 30-2.

If you do not explicitly specify a status in your annotation, Polyspace assumes that you have set a status of `No Action Planned`.

See Also

More About

- "Annotate Code and Hide Known or Acceptable Results" on page 30-2

Import Review Information from Existing Polyspace Access Projects

This topic describes how to import review information from previous results in Polyspace Access. For information on importing from results that are not uploaded to Polyspace Access, see “Import Review Information from Previous Polyspace Analysis”.

If you review findings in a Polyspace Access project and you reuse the same source code that contains those findings in another project, you can import the existing review information to the other project. You do not need to review the findings again in the other project. The project you import from is the source project. The project you import to is the target project.

For instance, suppose your team has reviewed all findings for the file `customClass.cpp` in the Polyspace Access project `components/oldProject (BF)`. If you use `customClass.cpp` in a different project, you can import review information such as **Status** and **Severity** from `components/oldProject (BF)` into the other project.

Note that when you upload a run to a project, Polyspace Access automatically imports the review information from previous runs of that project to the newly uploaded run.

These values are imported when you merge review information between projects:

- **Status**
- **Severity**
- **Assigned to**
- **Comments**
- **Ticket Key**

For Polyspace Code Prover results, the source keeps its review information if the color of the check changes.

Import Review Information from Source Project to Target Project in Polyspace Access

Import review information from a source project to a target project.

1. In the **Project Explorer**, right-click your target project and select **Import Reviews from Another Project**.
2. Locate and select the source project. You can only import from one source at a time.
3. Select one of these import policies:

   - **Write review when there is no review on target finding**

     Import non-empty source project review information only if all the review fields of the target finding are empty.
   - **Source reviews always replace target**

     Import non-empty source project review information and replace target fields, even if the target review already contains review information.

4 Click **OK**. A dialog box appears at the top of the **Dashboard** when the import begins and when it completes. Imports for larger projects can take several minutes to complete.

View and Select Imported Reviews

Click **Open Results** in the **Project Overview** dashboard to view the **Results List**. Project results with imported review information have an icon next to them in the **Family**column. The icon indicates the current state of the imported review. This table describes the states of imported reviews.

Icon	Status	Status Description
	Not Applied	Review information from the source project is not applied to the target project findings.
	Overwritten	Review information from the source project is applied to the target project finding. The source overwrites the original target review information.
	Written	Review information from the source project is applied to previously empty target project findings.

You can switch between original and imported review information. To decide what review information to use, view imported and original result information side-by-side. Use the **Imported Review Selection** window to view result information in this way. To access this pane:

- Right-click the result you want to review in the **Results List** .
- Select **Show Imported Review Selection**.

In the **Imported Review Selection** panel, three columns represent the **Review Fields**, the **Original Values**, and the **Imported Values**. A radio button next to **Original Values** and **Imported Values** enables you to quickly select which values to apply to the findings.

Confirm Imported Review Information

Imported review information is considered unconfirmed until you manually confirm it. You can confirm result information individually or as a group.

1 Expand the **Filters** list and select the **Unconfirmed Imports** filter.

2 Select a result. To select multiple results, click those results while holding the **Ctrl** key. To select a range of adjacent results, click the first and last result in that range while holding the **Shift** key.

3 Right-click the selected result, go to **Confirm imported reviews** in the context menu and choose one of these options.

Option	Option Description
Use original values	Keep the original review information in the target finding for all selected results. If this option is grayed out, the target finding already uses the original review information.
Use imported values	Apply the imported review information to the target finding for all selected results. If this option is grayed out, the target finding already uses the imported review information.
Confirm current selection	Confirm review information as it currently is set. If this option is grayed out, the target finding is already confirmed.

In some instances, it is useful to know the review history of a result. To open the **Review History** pane, from the **Result Details** pane, click the icon. The **Review History** shows information about changes to individual result details including the name of the editor and time of the edit. See "Review History in Polyspace Access Web Interface".

Import Review Information at the Command-Line

To import review information from an existing project to another project that reuses the code that contains the reviewed finding, use command `polyspace-access -import-reviews` and specify:

- The full path of the project that you want to import the review information from (source project).
- The full path of the project that you want to import the review information to (target project).
- Optionally, you can specify one of these import strategies:
  - `never-overwrite-target` (default) — If a review field in the target project is already filled, do not overwrite it with the content from the source project
  - `always-overwrite-target` — Always overwrite the content of the review fields in the target project, even if they are already filled.

For example, if you have already reviewed findings in project `public/example/branchA`, and you reuse the reviewed code in project `public/example/branchB`, use this command to import the review information from `branchA` to `branchB`:

```
polyspace-access $login -import-reviews public/example/branchA \
 -to-project-path public/example/branchB
```

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

After you complete the import, you might want to examine the result of the import operation. Use the `polyspace-access -export` command with option `-imported-reviews` and filter by one of these values:

- `Not applied` — Review information was imported from the source project but the review fields in the target project kept the original values.

- `Written` — The Review information from the source project was written to the target project only if the review fields in the target project were previously empty.
- `Overwritten` — The Review information from the source project was written to the target project even if the review fields in the target project were not previously empty.
- `Unconfirmed` — Use this filter to select findings that where the result of the import operation has not been confirmed by a reviewer. You confirm the result of the import operation in the Polyspace Access interface. See "Confirm Imported Review Information" on page 27-6 .

For example, to get a list of findings from the target project whose review information was overwritten, enter this command:

```
polyspace-access $login -export public/example/branchB \
 -imported-reviews Overwritten -output overWrittenReviews.txt
```

The command output tab-separated values (TSV) file `overWrittenReviews.txt` which contains only findings that had their review information overwritten in the target project.

See Also

`polyspace-access` | `polyspace-code-prover-server`

Related Examples

- "Address Results in Polyspace Access Through Bug Fixes or Justifications" on page 27-2

Manage Results

- “Manage Permissions and View Project Trends in Polyspace Access Web Interface” on page 28-2
- “Filter and Sort Results in Polyspace Access Web Interface” on page 28-8
- “Create Custom Filter Groups in Polyspace Access Web Interface” on page 28-13
- “Manage Software Quality Objectives in Polyspace Access” on page 28-15
- “Add Labels to Project Runs in Polyspace Access” on page 28-18
- “Prioritize Check Review in Polyspace Access Web Interface” on page 28-21
- “Compare Results in Polyspace Access Project to Previous Runs and View Trends” on page 28-23

Manage Permissions and View Project Trends in Polyspace Access Web Interface

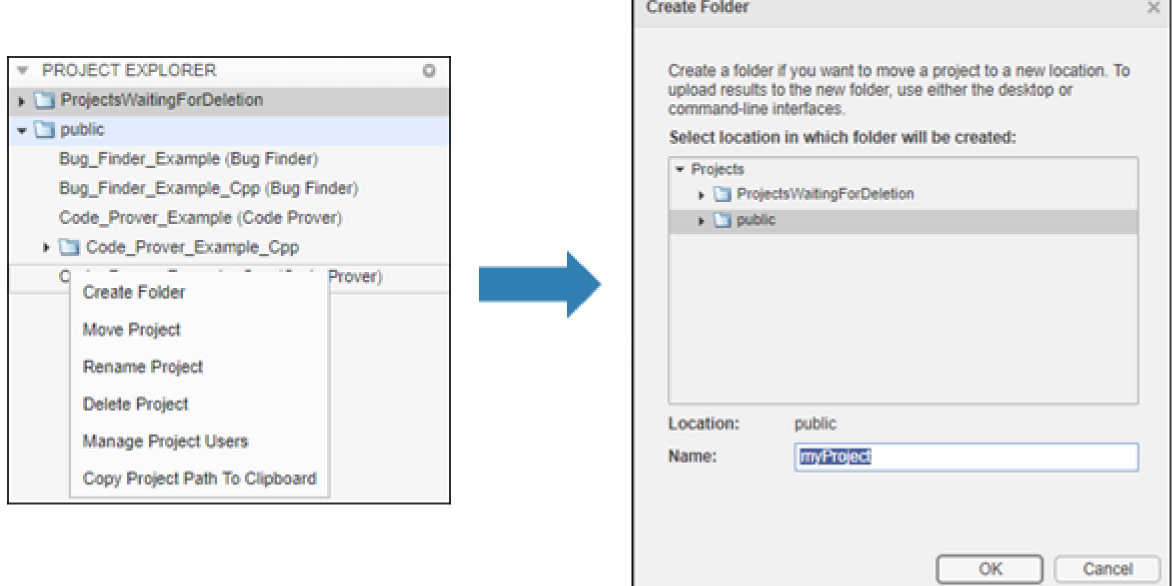
Before you start reviewing the overall quality of a project and investigating findings in your code, create project folders and set permissions to allow or restrict team members access to your projects.

Create a Project Folder

To facilitate the review process, create folders in Polyspace Access to group related results.

Create Folder from the Polyspace Access Interface

From the **Project Explorer** in the **DASHBOARD** perspective, select any existing folder or project and click **Create Folder** in the context menu. In the **Create Folder** window, click an existing folder to create a subfolder. To create a folder at the top of the **Project Explorer** hierarchy, click **Projects**.



Create Project Folder at Command Line

To create a folder in Polyspace Access from the DOS or UNIX command lines, use the `polyspace-access` binary. This binary is available under the *`polyspaceroot`*`/polyspace/bin` folder. The *`polyspaceroot`* folder is the Polyspace product installation folder, for example `C:\Program Files\Polyspace Server\R2023a`.

For instance, to create `myProject` under the folder `myRelease`, use this command:

```
polyspace-access -host hostName -port port -create-project myRelease/myProject
```

`hostName` and *`port`* correspond to the host name and port number that you specify in the URL of the Polyspace Access interface, for example `https://`*`hostName`*`:`*`port`*`/metrics/index.html`. If you are unsure about which host name and port number to use, contact your Polyspace Access

administrator. Depending on your configuration, you might also need to specify the `-protocol` option in the command.

Manage Project Permissions

To set permissions for folders or projects in Polyspace Access, assign roles to users or groups. The permissions that correspond to each role are listed in this table.

Role	Permission
Administrator	• Move, rename, or delete any folder or project and review their content. • Assign roles **Owner**, **Contributor**, or **Forbidden** to other users or groups. • View and manage contents of **ProjectsWaitingForDeletion** folder. See "Delete Project Runs or Entire Projects". To set a user as **Administrator**, see "Configure User Manager". You cannot move a folder or project to a new location if a folder or project with the same name already exists at that location.
Owner	• Move, rename, or delete folders or projects that you own and review their content. • Assign roles **Contributor** or **Forbidden** to other users or groups. You are the owner of folders that you create and of project results that you upload. You cannot move a folder or project to a new location if a folder or project with the same name already exists at that location.
Contributor	• Review content of folders or projects for which you are a contributor. • See the roles of other users and groups for the project.
Forbidden	No access to the specified folder or project. Set this role to restrict the access of a user or group if: • The user or group inherits access from a parent folder. • The user or group inherits access from a parent group. .

- The user or group roles that you assign for a project folder apply to all the projects and subfolders under that folder. You can also set different user or group roles for each project or subfolder. For instance, you can assign user `jsmith` as a contributor for folder `myRelease`, and then restrict the access of `jsmith` to subfolder `myRelease/update1`.
- Only **Administrator** or **Owner** roles can allow or restrict the access of other team members or groups to a project or folder.
- Only **Administrator** roles can assign other users or groups as owners of a project or folder.
- Unless you explicitly set a user or group role for a project, the user or group inherit the role of their parent group for that project. For instance, if user `jsmith` is not assigned any role for folder `myRelease`, and `jsmith` is a member of a group that is a contributor for folder `myRelease`, then `jsmith` is also a contributor to folder `myRelease`.

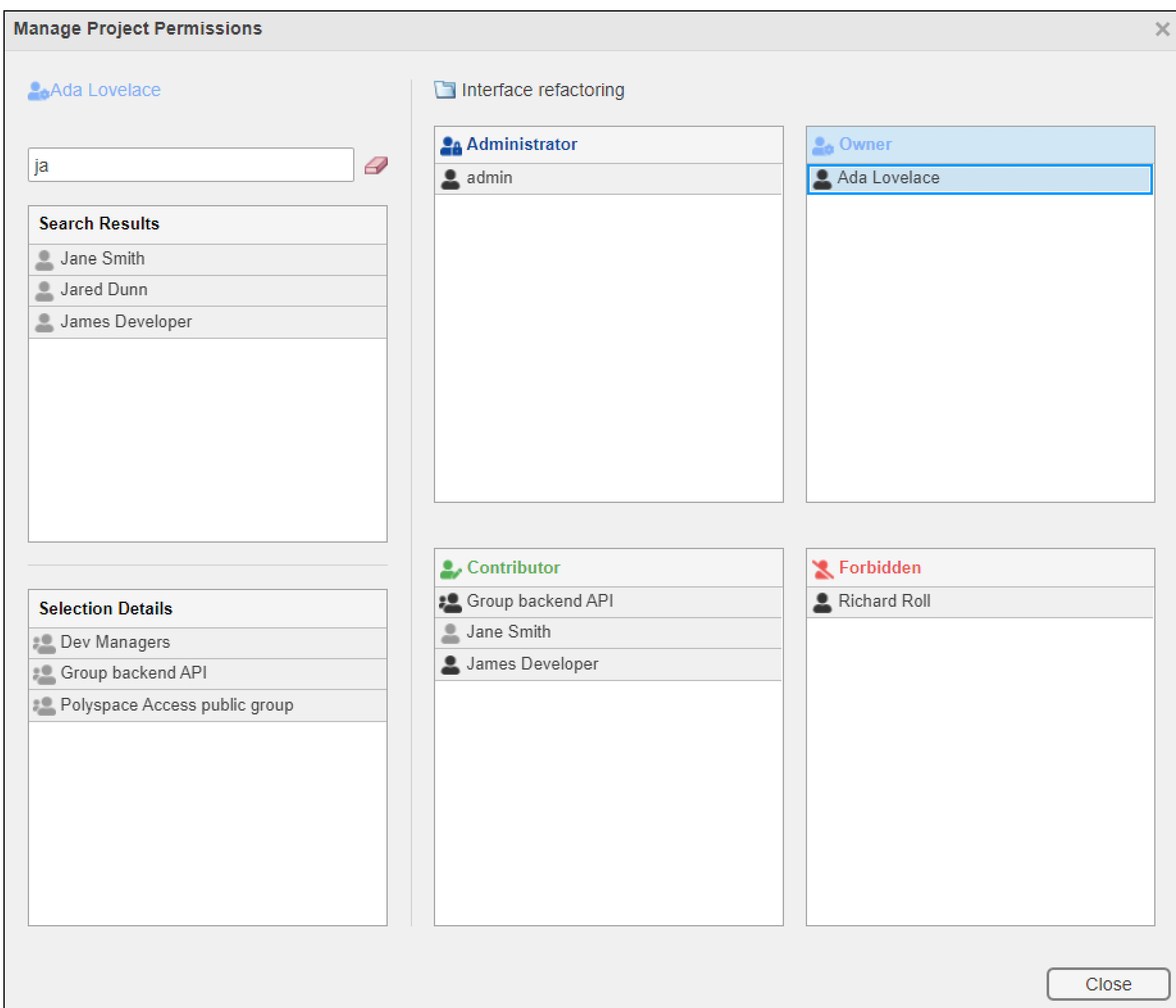
By default, all users are members of the **Polyspace Access public group** and all users inherit the role of that group (**Contributor**) for the **public** folder. You cannot change the permissions for the **public** folder, but you can change permissions for subfolders or projects inside the **public** folder.

Manage Permissions in Polyspace Access Web Interface

From the **Project Explorer** in the **DASHBOARD** perspective, select any existing folder or project and click **Manage Project Permissions** in the context menu.

The **Manage Project Permissions** interface opens for the selected project.

- To assign or unassign roles, right-click a user or group in any of the panes.
- Place your cursor over a user or group in any of the panes to see a tooltip that has information about the user or group role for the selected project, and whether that role is inherited from a parent project or parent group.



This table provides additional information about the different panes in the **Manage Project Permissions** interface.

Pane	Description
Search Results	To view a list of user or groups that match your search string, type the user name or group name in the search bar.
Selection Details	From any of the other panes, click a user to view the groups that the user belongs to in this pane. If you click a group, this pane shows only the direct descendant members of the group. For instance, if group `nestedGroup` is a member of group `parentGroup`, when you click `parentGroup`, you do not see the members of `nestedGroup` in this pane.
Administrator **Owner** **Contributor** **Forbidden**	• View list of users or groups that have a role assigned for the project. • If the role is assigned explicitly on the selected project, the user or group icon is black, for instance . • If the role is inherited from a parent folder, the user or group icon is gray, for instance . • These four panes do not show users or groups that inherit their role from a parent group. For instance, if group `Contractors` is assigned as a contributor to the project, the members of this group are not listed in the **Contributor** pane.

The list of Polyspace Access users and groups (identities) is populated from the **User Manager** database. If an identity is removed from this database and the identity was assigned a role explicitly on at least one Polyspace Access project, that identity is highlighted in red in the **Manage Project Permissions** interface and is listed by ID instead of display name, for instance `jsmith`, instead of `John Smith`. A role is not explicitly assigned if it is inherited from a parent group or a parent project folder.

Contact a Polyspace administrator to remove that identity from the Polyspace Access. See “Update List of Polyspace Access Users and Groups”.

Identities that are deleted from the **User Manager** database and that do not have roles explicitly assigned to them are removed from Polyspace Access when you refresh your web browser.

Manage Permissions at Command Line

To manage access to uploaded results from the DOS or UNIX command lines, use the `polyspace-access` binary. This binary is available under the *`polyspaceroot`*`/polyspace/bin` folder. The *`polyspaceroot`* folder is the Polyspace product installation folder, for example `C:\Program Files\Polyspace Server\R2023a`.

For instance, to assign `jsmith` as **Contributor** for project `myProject`, use this command:

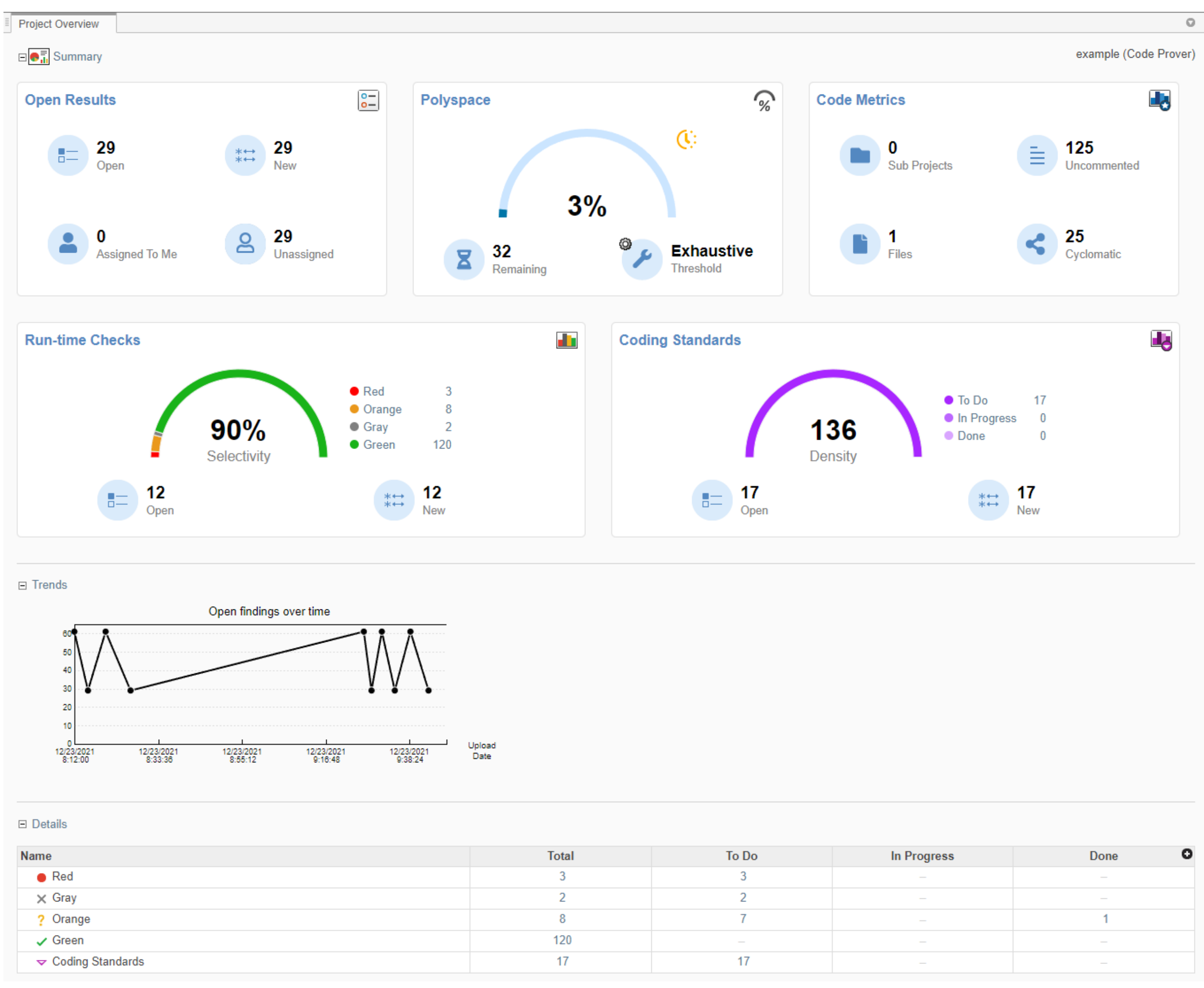
```
polyspace-access -host hostName ^
-set-role contributor -user jsmith ^
-project-path myProject
```

`hostName` and *`port`* correspond to the host name and port number that you specify in the URL of the Polyspace Access interface, for example `https://`*`hostName`*`:`*`port`*`/metrics/index.html`. If you

are unsure about which host name and port number to use, contact your Polyspace Access administrator. Depending on your configuration, you might also need to specify the `-protocol` option in the command.

You cannot assign the **Administrator** role to a user at the command line.

View Project Trends



In the **DASHBOARD** perspective, select the project that you want to investigate from the **PROJECT BROWSER**.

If you select a folder that includes multiple projects, the dashboard displays an aggregate of results for all the projects that you have permission to view. If the folder contains a project for which you are not an **Administrator**, **Owner**, or **Contributor**, results for that project are not included in the aggregate calculation.

In the **Project Overview** dashboard, you see a summary of **Open Issues**, including the number of **New** results since the previous analysis run and the number of results that are **Unassigned**.

Other cards provide statistics for each family of findings. The **Run-time Checks** card shows the **Selectivity**, that is, the percentage of all findings that are green. When you enable the calculation of code metrics in your analysis, the **Defects** and **Coding Standards** cards show the **Density**, the number of findings per thousand lines of code without comments.

In the **Details** section, you see the review progress for each family of results. The results are classified as:

- **To Do** — Findings with a status of `Unreviewed` that need to be addressed with a fix or a justification.
- **In Progress** — Findings with a status of `To fix`, `To investigate`, or `Other` that need to be addressed with a fix or a justification.
- **Done** — Findings with a status of `Justified`, `No action planned`, or `Not a defect`.

Note Green run-time checks, green shared variables, non-shared variables, and code metrics do not need to be addressed or justified. These findings do not count toward the number of findings that are **To Do**, **In Progress**, and **Done**.

If the number of open issues increases, open additional dashboards by using the buttons in the **DASHBOARDS** section of the toolstrip. Each button opens a dashboard for a family of findings, for instance **Defects**. To determine the root cause of the increase, use the information on these dashboards. Once you determine the set of findings that you want your team to focus on, open the **REVIEW** perspective to start managing the results. See “Manage Results”.

See Also

More About

- “Upload Results to Polyspace Access” on page 2-28

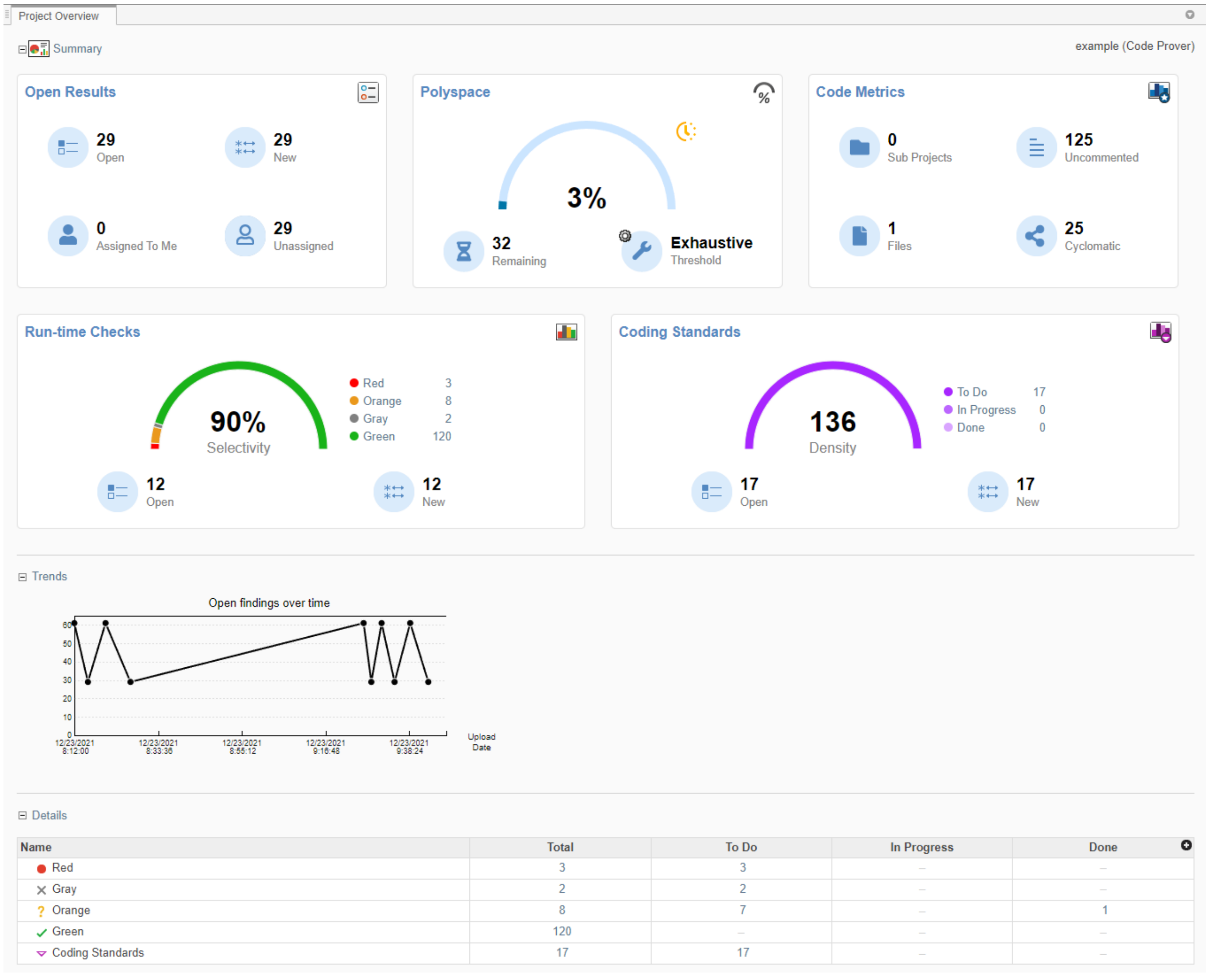
Filter and Sort Results in Polyspace Access Web Interface

This topic describes how to filter, sort, and otherwise manage results in the Polyspace Access web interface. For a similar workflow in the user interface of the Polyspace desktop products, see “Filter and Group Results in Polyspace Desktop User Interface” on page 24-2.

When you open the results of a Polyspace analysis in the **DASHBOARD** view of Polyspace Access, you see statistics about your project in the **Project Overview** dashboard. The statistics cover findings for:

- Bug Finder “Defects”.
- Code Prover “Run-Time Checks”.
- “Coding Standards” violations.
- “Code Metrics” and “Evaluate Polyspace Code Prover Results Against Software Quality Objectives” on page 31-2 compliance.

To organize your review, you can narrow down the list or group results by file or result type.



Some of the ways you can use filtering are:

- You can display only certain types of defects or run-time checks.

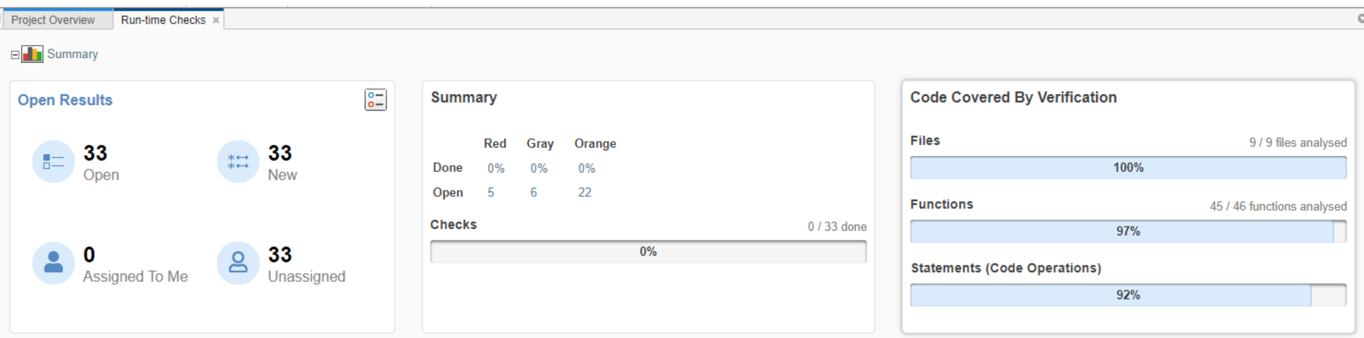
  For instance, for a Bug Finder analysis, you can display only high-impact defects. See “Classification of Defects by Impact”.
- You can display only new results found since the last analysis or since a previous analysis. See “Compare Results in Polyspace Access Project to Previous Runs and View Trends”.
- You can display only the results that you have not justified. Results that are not justified are considered **Open**. They are results with status `Unreviewed`, `To Investigate`, `To Fix`, or `Other`.

  For information on justification, see “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2 .
- You can display only results that you still need to address to reach a **Quality Objectives** threshold.

Filter Results

You can filter results by drilling down on a set of results in a dashboard, or directly in the **Results List** pane by using the **REVIEW** toolstrip filters.

Filter Using Dashboards

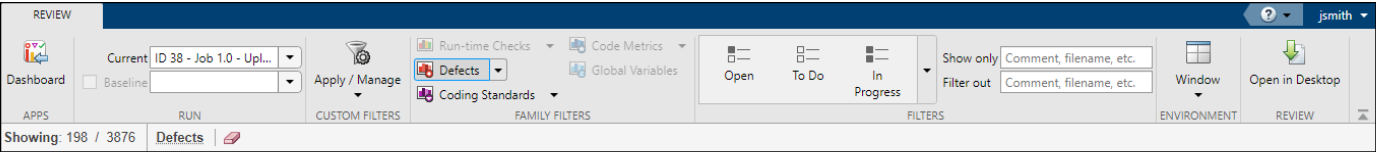


In the **DASHBOARD** view, you can:

- Click a section of a pie chart or a pie chart legend on the **Project Overview** dashboard to see the corresponding set of results.
- Open dashboards for different families of results, then click a number to open a list filtered to the corresponding set of results. For instance:
  - To see only high-impact defects that are still **Open** in a Bug Finder analysis, click the corresponding number in the **Summary** section of the **Defects** dashboard. **Open** results have status `Unreviewed`, `To Investigate`, `To Fix`, or `Other`.
  - To see only red checks that are **Done** in a Code Prover analysis, click the corresponding number in the **Summary** section of the **Run-time Checks** dashboard. **Done** results have status `Justified`, `No Action Planned`, or `Not A Defect`.
  - To see violations of the MISRAC C:2012 coding standards in a particular file, use the table in the **Details** section of the **MISRA C:2012** dashboard.
- Compare the **Current** run to an earlier **Baseline** run and review **New** or **Unresolved** findings. See "Compare Results in Polyspace Access Project to Previous Runs and View Trends".

If you select a folder that contains multiple projects in the **Project Explorer**, the dashboards display an aggregate of results for all the projects. Most of the fields in the dashboard are not clickable when you look at the statistics for multiple projects.

Filter Using REVIEW Toolstrip



In the **REVIEW** view, you can filter results by families of Polyspace results (**FAMILY FILTERS**), or by result review progress (**FILTERS**). For instance:

- To see Bug Finder defects only, select the **Defects** filter in the **FAMILY FILTERS** group.
- To see only results that are not justified, select the **Open** filter in the **FILTERS** group.

The filter bar underneath the toolstrip shows how many findings are displayed out of the total findings, along with which filters are currently applied.

Note If you are reviewing a filtered list of open results and you add an SQO filter, the number of filtered results might increase. This can happen when your project has code metrics with a status of FAIL. The SQO filter adds the failing code metric results to the list of results.

The buttons in the **FILTERS** section of the toolstrip are global. They apply to all families of findings.

To filter results by specific content, such as a function name, use the **Show only** or **Filter out** text filters. These filters match the text you enter against the content of all the columns in the "Results List in Polyspace Access Web Interface" on page 26-17. For instance, if you enter `foo` in the **Filter out** filter, the **Results List** hides all the results that contain `foo` in any of the **Results List** columns.

You can also filter results by right-clicking the content of a column in the **Results List**. This action is equivalent to entering the content directly in the **Show only** or **Filter out** text filters. For instance, if you right-click `foo` in the **Function** column, the filter applies to all results that contain `foo` in any of the **Results List** columns.

Filters you apply do not carry over to the next analysis.

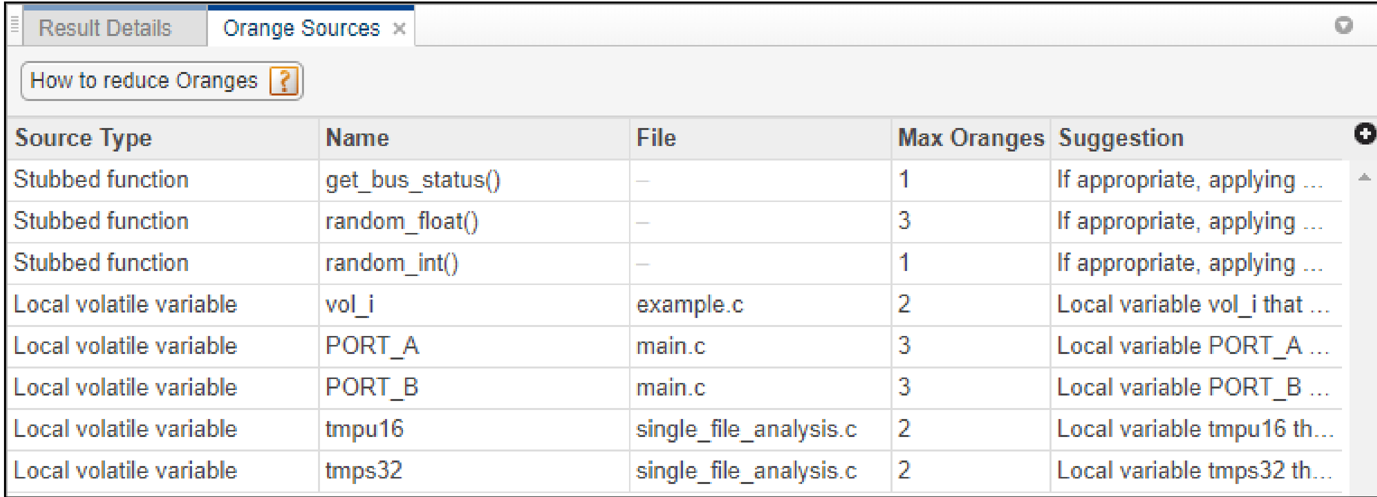
Filter Using Orange Sources

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, in the **REVIEW** view, select **Window > Orange Sources**. You see the list of orange sources. Select an orange source to see all orange checks coming from this source.



Source Type	Name	File	Max Oranges	Suggestion
Stubbed function	get_bus_status()	–	1	If appropriate, applying ...
Stubbed function	random_float()	–	3	If appropriate, applying ...
Stubbed function	random_int()	–	1	If appropriate, applying ...
Local volatile variable	vol_i	example.c	2	Local variable vol_i that ...
Local volatile variable	PORT_A	main.c	3	Local variable PORT_A ...
Local volatile variable	PORT_B	main.c	3	Local variable PORT_B ...
Local volatile variable	tmpu16	single_file_analysis.c	2	Local variable tmpu16 th...
Local volatile variable	tmps32	single_file_analysis.c	2	Local variable tmps32 th...

See Also

More About

- “Prioritize Check Review in Polyspace Access Web Interface” on page 28-21

Create Custom Filter Groups in Polyspace Access Web Interface

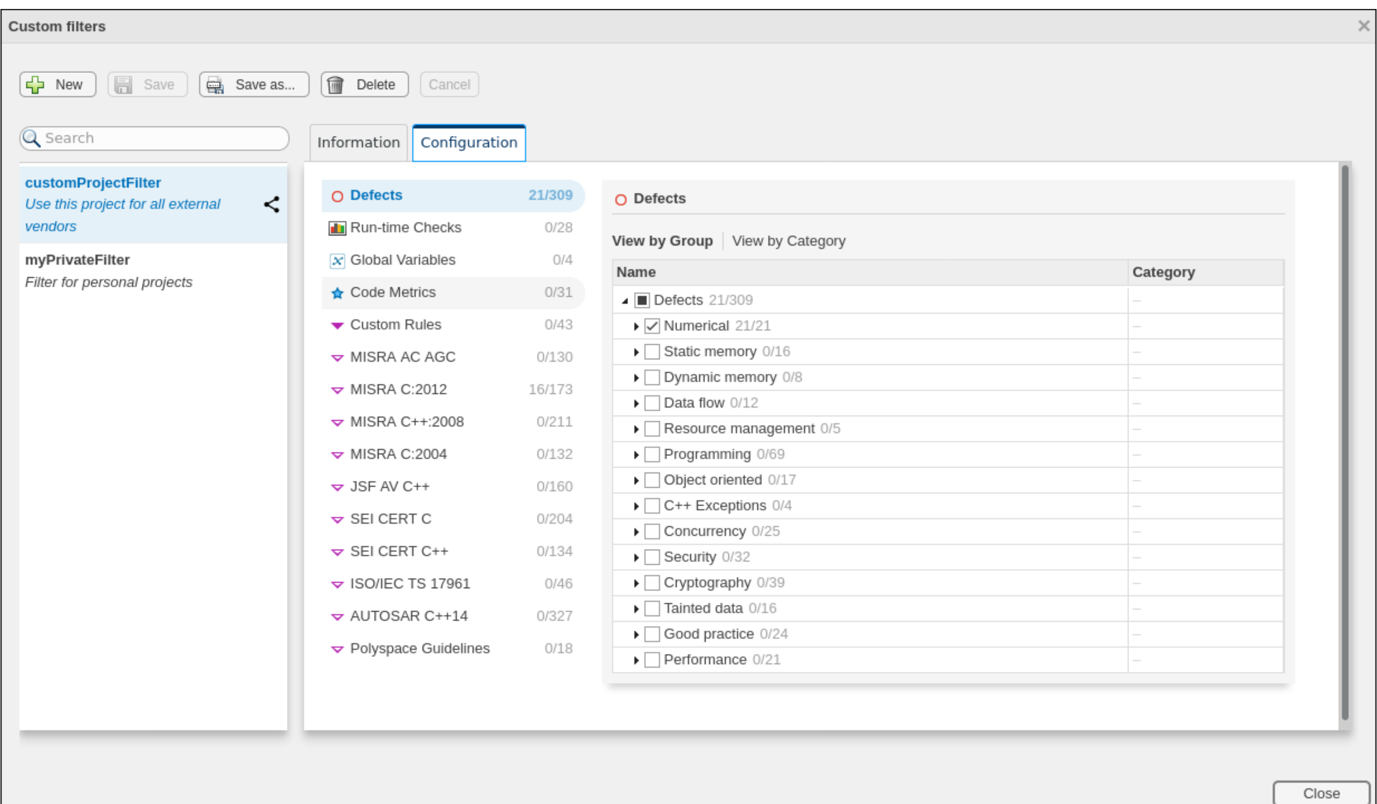
When you review results in the **Results List**, you can apply filters from the **FAMILY FILTERS** section of the toolstrip to focus your review on specific Polyspace families of results, such as:

- Bug Finder “Defects”.
- Code Prover “Run-Time Checks” and “Global Variables”.
- “Coding Standards” violations.
- “Code Metrics”.

Define custom filters to narrow the scope of your review to only findings that are relevant to your project or organization. For instance, you might be interested in reviewing only **Numerical** Bug Finder defects and violations of **Mandatory** MISRA C:2012 rules.

Once you define custom filters, you can share those filters with other Polyspace Access users to ensure consistent review scopes across your projects or organization.

To create or edit a custom filter, click **Apply/Manage** > **Manage filters**.



To create a new filter, in the **Custom filters** window, click **New** and then enter the filter name in the **New Custom Filter** pop-up window. You can optionally provide a description and enable the **Shared filter** checkbox to share the filter with other Polyspace Access users.

By default, custom filters are private and can be viewed only by the user who creates the filter. A private filter can be edited only by the user who creates that filter. A shared filter can be edited by the user who creates the filter or by a user with the role of **Administrator**.

To make changes to a filter name, description, or to enable or disable filter sharing, go to the **Information** tab.

To edit the filter selection, on the **Configuration** tab, click a Polyspace results family, for instance MISRA C:2012, and then select a node or expand the node to select individual results. For each family of results, you can view the nodes by group or by category when available.

To save your changes, click **Save** or **Save as** to save your edits in as new custom filter.

Apply custom filters by selecting the appropriate filter from **Apply/Manage > Private filters** or **Apply/Manage > Shared filters**. You can apply more than one custom filter, including combinations of private and shared filters.

Custom filters do not apply to the **DASHBOARD** view.

See Also

Related Examples

- “Filter Results”
- “Customize Software Quality Objectives”

Manage Software Quality Objectives in Polyspace Access

To monitor the quality of your code against predefined on page 31-2 software quality thresholds or user-defined thresholds, use the **Quality Objectives** dashboard. See “Quality Objectives Dashboard in Polyspace Access” on page 26-12.

The first time that you upload results to a new project, Polyspace Access assigns the default **Polyspace Software Quality Objectives** definition to that project. To create custom software quality objective (SQO) definitions, see “Customize Software Quality Objectives” on page 26-14.

You can manage the SQO of a project from the user interface or at the command line.

After you assign an SQO definition, you see the label (not computed) on the **Quality Objectives** card and dashboard until the project statistics are recalculated.

The SQO statistics for a project are recalculated when:

- You upload a new run for the project.
- You select a finding and make a change to any of the fields in the **Result Details** pane.

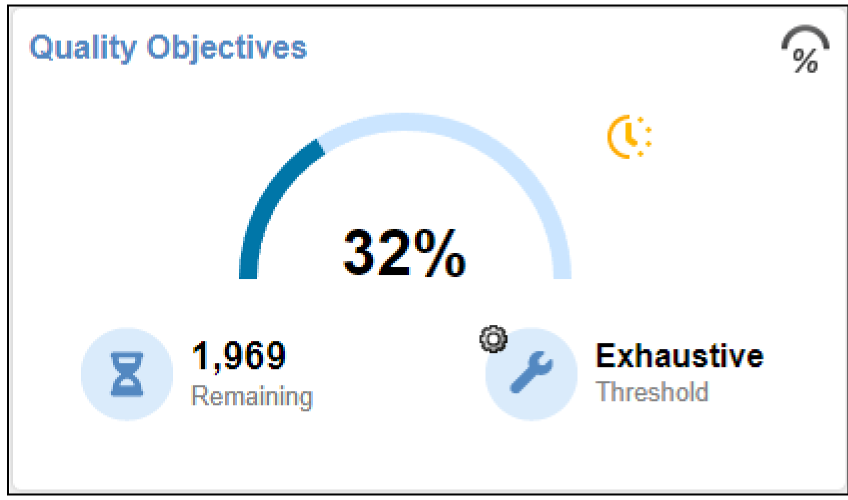
The SQO definition that you assign to a project applies only to runs that you upload to the project after assigning the definition.

Tip When the **Quality Objectives** settings and the calculated statistics for a project are out of sync, the **Quality Objectives** dashboard displays a warning .

If you delete an SQO definition, Polyspace Access assigns the **Polyspace Software Quality Objectives** to all the projects to which the deleted definition was assigned.

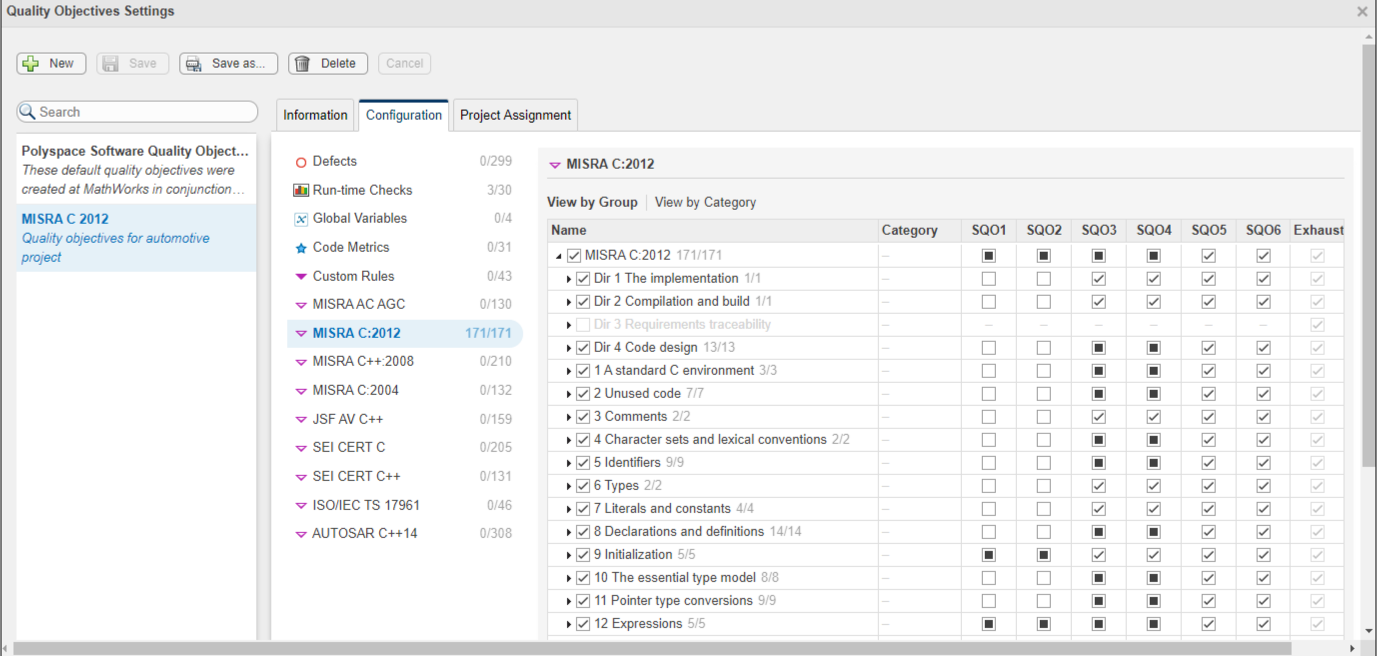
Manage SQOs in the User Interface

To assign an SQO definition or level to a project, right-click a project in the **Project Explorer** or click on the **Quality Objectives** card or the **Quality Objectives** dashboard.



If you make changes to a quality objectives definition that applies to multiple projects, Polyspace Access displays a warning with a link to the **Project Assignment** tab on the **Quality Objectives Settings** window. Open the tab to determine which projects are affected by your changes and inform users that are contributors to those projects of the changes to the quality objectives definition.

To view which projects an SQO definition is assigned to, go to the **Project Assignment** tab in the **Quality Objectives Settings**.



Manage SQOs at the Command Line

To manage SQOs from the command line, use the `polyspace-access` command. In the following examples, *`$LOGIN`* is a variable that stores the login credentials and other connection information. To configure this variable, see “Encrypt Password and Store Login Options in a Variable”.

You can:

- Assign an SQO level, and optionally an SQO definition to a project. For instance, to assign level 3 of SQO definition `My Custom SQO` to project `myProject` with project path `public/examples/myProject`, enter this command:

  ```
  polyspace-access -set-sqo public/examples/myProject -level 3 -name "My Custom SQO" $LOGIN
  ```

  Option `-level` is mandatory and can be any value from 1 to 6 or "exhaustive", while option `-name` is optional.

  If you do not use `-name`, the level that you specify is applied for the currently assigned SQO definition.
- View the currently assigned SQO definition and SQO level for a project. For instance, to view the assigned SQO level and definition for project `myProject` with project path `public/examples/myProject`, enter this command:

  ```
  polyspace-access -get-sqo public/examples/myProject $LOGIN
  ```

The command outputs the SQO name and level in this format:

```
Current Quality Objectives: NAME My Custom SQO LEVEL SQO-3
```

- View a list of all currently available SQO definitions. Enter this command:

```
polyspace-access -list-sqo $LOGIN
```

See Also

More About

- "Quality Objectives Dashboard in Polyspace Access" on page 26-12
- "Evaluate Polyspace Code Prover Results Against Software Quality Objectives" on page 31-2
- "Code Metrics"

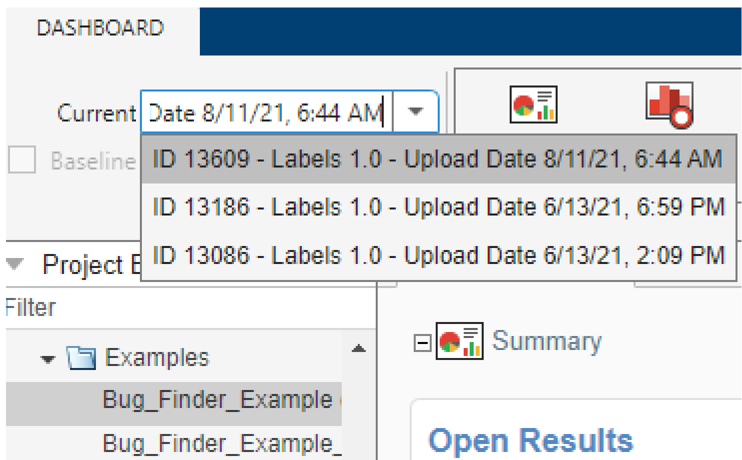
Add Labels to Project Runs in Polyspace Access

To help identify project runs uploaded to Polyspace Access, you can assign custom labels to runs. Custom labels are in addition to the unique run IDs that Polyspace Access assigns to each run.

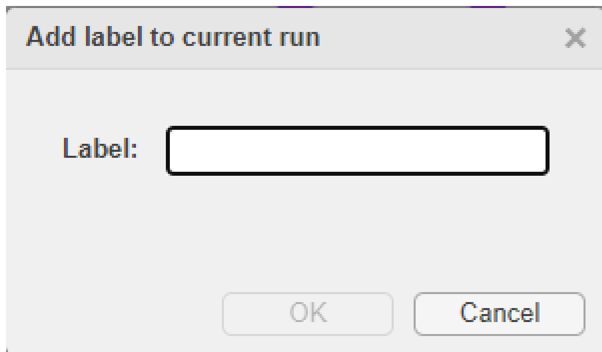
You can assign labels from the user interface through the **Project Details** pane on the **Dashboard**, or from the command line.

Manage Labels in the User Interface

To add a label to a run, first select a project in the **Project Explorer**. Select the run that you want to add a label to by using the **Current** drop-down list.

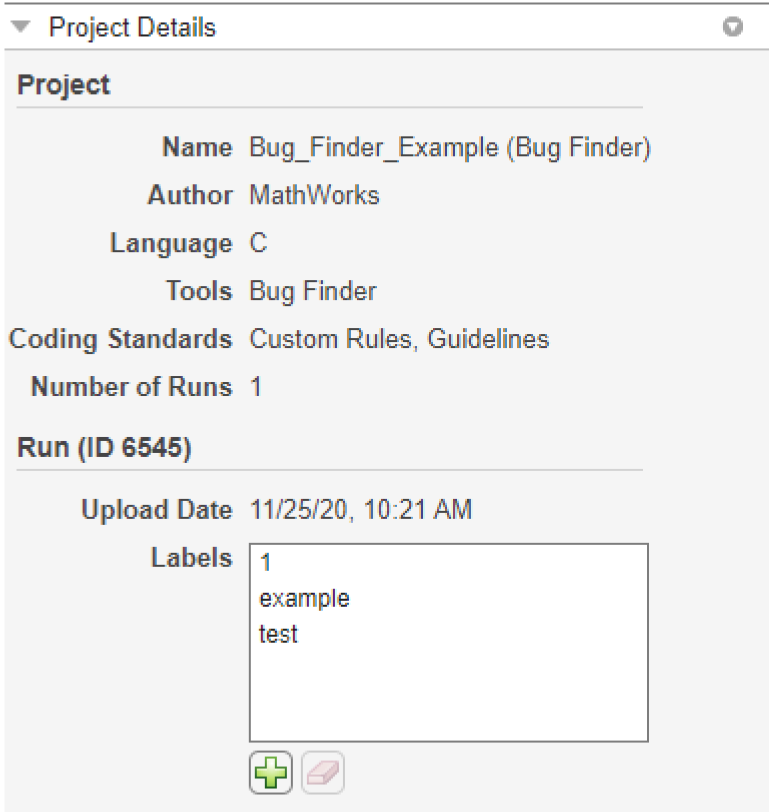


In the **Project Details** pane, in the **Labels** box under the **Run** section, click the icon. In the **Add label to current run** box, enter the label name to assign to the run, and then click **OK**.



Labels are sorted in alphabetical order. There is no limit on how many labels you can assign to a single run.

To delete a label, select the label and click the icon. You can select multiple labels to delete them simultaneously.



Manage Labels at the Command Line

To add a label to a run from the command line, use the `polyspace-access` command with the `-add-label` option.

For instance, suppose that you use a continuous integration tool like Jenkins and that you want to associate the Jenkins build number with the project run that you upload to Polyspace Access. The following steps show how to extract the run ID of the uploaded project run and add a label to that run by using Bash commands:

1 Store the output of the `polyspace-access -upload` command to a file `out.txt` which you can then parse to extract the run ID of the uploaded run.

```
polyspace-access $login -upload results/Folder/Path -parent-project myProject -output out.txt
runID=$(grep -oP '(?<=RUN_ID )\d+' out.txt)
```

Here:

- The `grep` expression extracts the digits after the string "`RUN_ID `" in file `out.txt`. The content of that file looks similar to this:

```
Upload with IMPORT_ID 1640263976711_d8b0fc8b-edfe-41c4-b718-6fd4b930e910.zip
Upload successful for RUN_ID 14970 and PROJECT_ID 5145
```

- `$login` is a variable that stores the login credentials and other connection information. To configure this variable, see "Encrypt Password and Store Login Options in a Variable".

If you use DOS commands, you can extract the run ID by using a `for` loop:

```
for /f "skip=1 tokens=5" %i in (out.txt) do set runID=%i
```

The loop skips the first line of the file and extracts the fifth space-delimited element (token) in the second line.

2. Add the Jenkins build number as a label to the project run that you uploaded. You can obtain the Jenkins build number for the Jenkins environment variable `BUILD_NUMBER`. Run this command:

   ```
   polyspace-access $login -add-label $BUILD_NUMBER -run-id $runID
   ```

   To add additional labels to the project run, execute the command again for each label. You cannot specify the `-add-label` option more than once each time you execute the command.

   If the label that you specify for addition to a project run is already assigned to that run, the command is ignored.

To remove a label, use the `polyspace-acess` command with the `-remove-label` option. For example, to remove the label you added in step 2, use this command:

```
polyspace-access $login -remove-label $BUILD_NUMBER -run-id $runID
```

If the label that you specify for removal from a project run does not match any of the labels assigned to that run, the command is ignored.

Prioritize Check Review in Polyspace Access Web Interface

This example shows how to prioritize your check review. Try the following approach. You can also develop your own procedure for organizing your orange check review.

Tip For easier review, run Polyspace Bug Finder on your source code first. Once you address the defects that Polyspace Bug Finder finds, run Polyspace Code Prover on your code.

Before beginning your check review, you can check the following:

- See the **Run Log** by going to **Window** > **Run Log** in the **REVIEW** view. Use `CTRL-F` to search the log for warning and error messages, or the string `failed compilation`. If there are warnings or errors, or files failed to compile, identify why Polyspace could not analyze all of your source files.

  To check for some common *Reasons for Unchecked Code*, see the documentation for Polyspace Code Prover.
- See if you have used the right configuration. The configuration options are listed in the **Run Log** under the strings `Options used with Verifier:` and `User:`.

  Sometimes, especially if you are switching between multiple configurations, you can accidentally use the wrong configuration for the verification.

1. From the **Project Overview** dashboard, click the number next to **Open** on the **Run-time Checks** card.

   This action opens the **Results List** pane with only unreviewed red, gray and orange checks. You can also filter for these results from the toolstrip in the **REVIEW** view by clicking **Run-time Checks** and **To Do**.
2. Select and review the first check.

   For more information, see “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2.

   Continue going through the list until you have reviewed all of the checks.
3. Before reviewing orange checks, review red and gray checks.
4. Prioritize your orange check review by:

   - For easier review, begin your orange check review from files with *fewer* orange checks.

     To sort files by number of orange checks, in the **Details** section of the **Run-time Checks** dashboard, click **View by File**, then click the head of the **Orange** column to sort it. Click an entry from this column to open the corresponding list of orange checks.
   - Check type: Review orange checks in the following order. Checks are more difficult to review as you go down this order.

Review Order	Checks
First	• `Out of bounds array index` • `Non-initialized local variable` • `Division by zero` • `Invalid shift operations`
Second	• `Overflow` • `Illegally dereferenced pointer`
Third	Remaining checks

- Orange check sources: Review all orange checks caused by a single variable or function. Orange checks often arise from variables whose values cannot be determined from the code or functions that are not defined.

  To review the sources of orange checks, select an orange check from the **Results List** pane then click [?] in the **Results Details** pane. You can also open the **Orange Sources** pane by going to **Window** > **Orange Sources**. For more information, see "Filter Using Orange Sources".
- Result details: Review all results that originate from the same cause. Sometimes, the **Detail** column on the **Results List** pane shows additional information about a result. For instance, if multiple issues trigger the same coding rule violation, this column shows the issue. Click the column header so that results that originate from the same type of issue are grouped together. Review the results in one go.

5. To see what percentage of checks you have justified, go to the **DASHBOARD** view and see the **Summary** section of the **Run-time Checks** dashboard.

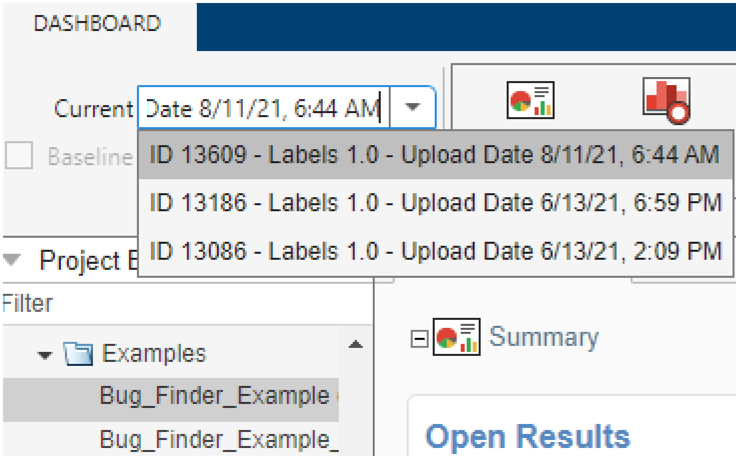
See Also

Related Examples

- "Filter and Sort Results in Polyspace Access Web Interface"

Compare Results in Polyspace Access Project to Previous Runs and View Trends

When you open Polyspace analysis results in the Polyspace Access **DASHBOARD** or **REVIEW**, you see a snapshot of the most recent run that was uploaded to the project. To view a snapshot from an earlier run, select that run from the **Current** run drop-down list.



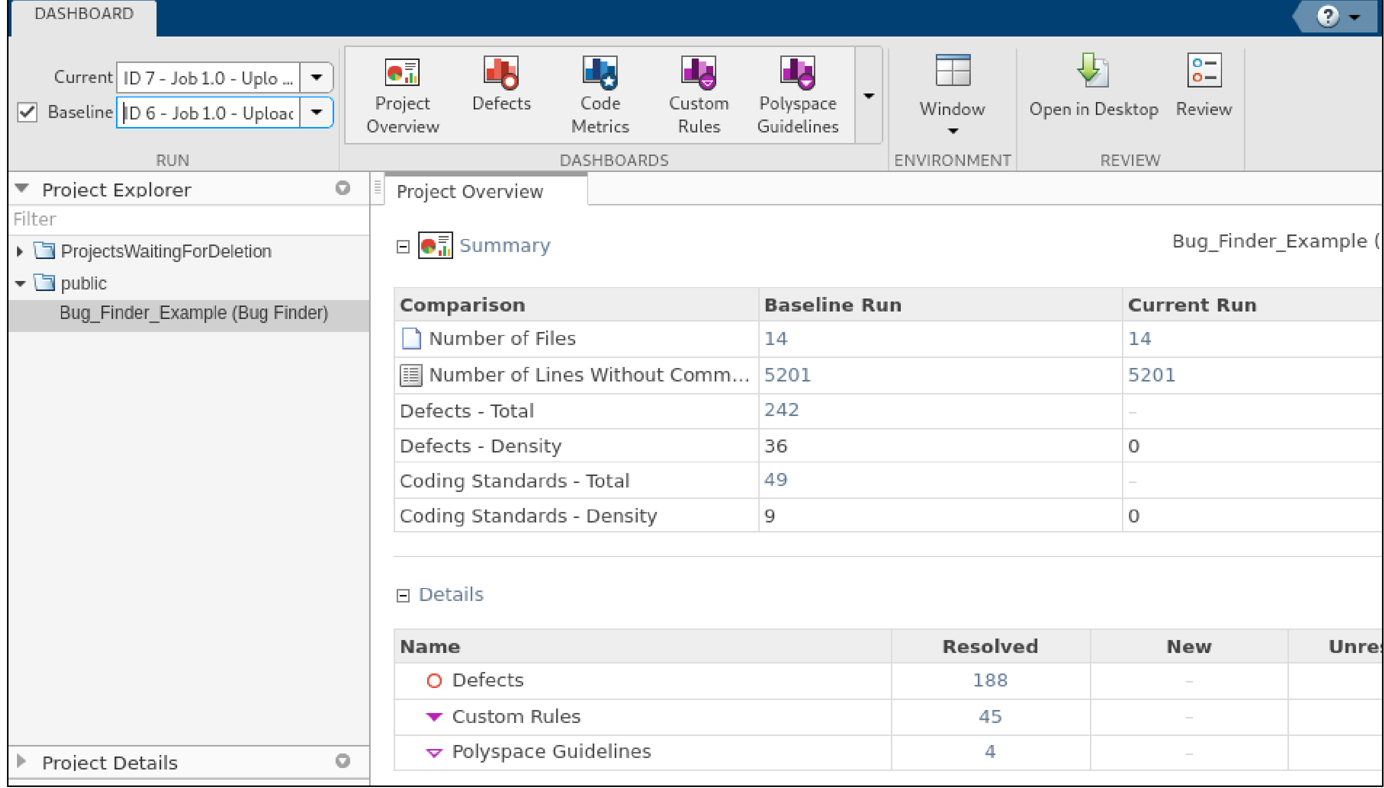
Select a previous run to see the state of your project from a few submissions ago. For instance, you might want to investigate a spike in findings in a previous version of your project. When you view an older project run in the **DASHBOARD** or **REVIEW** views, all the information for the currently selected run is displayed, except:

- The **Quality Objectives** settings and the **Review History** pane show the same information no matter which run you select.
- You cannot edit the **Result Details** fields if the selected run is not the latest run.

If you share a finding URL from an older run, the Polyspace Access interface opens that finding in the most recent version of the project. If the finding is not present in the most recent run, through the interface, you can open the finding in the older run.

Comparison Mode in the Polyspace Access Interface

To compare two runs in from the same project in the Polyspace Access interface, on the toolstrip, select a **Current** run and a **Baseline** run. Check that the **Baseline** checkbox is enabled. You can compare current runs to only older baseline runs.



In the **DASHBOARD** view, the comparison shows a summary of statistics for each run and details of the number of findings that are:

- **Resolved** — Findings from the baseline run that are **Done** in the current run, or findings that are not in the current run because they are **Fixed**.
  - Findings are **Done** if they have a status of `Justified`, `No Action Planned`, or `Not A Defect`.
  - Findings are **Fixed** if they are fixed in the source code or the source code containing the finding is deleted or no longer part of the analysis.
- **New** — Findings that are in the current run but not in the baseline run.
- **Unresolved** — Findings that are in the baseline run and the current run.

The comparison mode is not available for the **Code Metrics** and **Quality Objectives** dashboards.

Click a cell in the **Details** table to open the corresponding results in the **Results List**.

- The total number of findings displayed in the **Results List** corresponds to the findings from the **Current** run and the findings from the **Baseline** run that are **Fixed** in the **Current** run.
- If a finding is **Resolved**, the interface displays the **Source Code** and **Result Details** information from the **Baseline** run.

In the **REVIEW** view, in addition to **Resolved**, **New**, and **Unresolved**, you can filter findings by **Fixed**. These findings are no longer in the current run because they are fixed, or the source code containing the findings is deleted or no longer part of the analysis.

To turn off the comparison mode, deselect the **Baseline** checkbox or select `None` in the **Baseline** drop-down list.

Comparison Mode at the Command Line

To compare two runs from the same project at the command-line, use the `polyspace-access -export` command and specify the run ID of a current run, the run ID of an earlier run that you use as a baseline, and the resolution type that you want to use for the comparison.

When you specify a baseline to compare with the current run, the run ID that you specify for the baseline run must exist and must point to a run in the same project as the current run.

The command generates a file with a list of findings filtered by one of these resolution types:

- `New` — Findings that are in the current run but not in the baseline run.
- `Fixed` — Findings that are fixed in the current run, either because the finding was fixed in the source code, or because the source code containing the finding is deleted or no longer part of the analysis.
- `Unresolved` — Findings from the baseline run that are still present in the current run.
- `Resolved` — Findings that are `Fixed` in the current run or findings with a status of `Justified`, `No Action Planned`, or `Not A Defect` in the current run.

For example, to compare the latest run of project `public/Bug_Finder_Example(Bug Finder)` to an earlier run:

1. Use the `polyspace-access -list-runs` to obtain the run IDs of the runs that you want to compare:

   ```
   polyspace-access $login -list-runs "public/Bug_Finder_Example(Bug Finder)"

   Connecting to https://example-access-server:9443
   Connecting as jsmith
   PROJECT_PATH "public/Bug_Finder_Example(Bug Finder)" RUN_ID 28
   PROJECT_PATH "public/Bug_Finder_Example(Bug Finder)" RUN_ID 29
   PROJECT_PATH "public/Bug_Finder_Example(Bug Finder)" RUN_ID 30
   PROJECT_PATH "public/Bug_Finder_Example(Bug Finder)" RUN_ID 124
   PROJECT_PATH "public/Bug_Finder_Example(Bug Finder)" RUN_ID 125
   ```

   Here, `$login` is a variable that stores the login credentials and other connection information. To configure this variable, see “Encrypt Password and Store Login Options in a Variable”.
2. Use the command `polyspace-access -export` and specify:

   - The run ID of a current run.
   - The run ID of an earlier run that you use as a baseline.
   - The resolution type that you want to use a as a filter.

   For instance, to compare the last run (run ID 125) to the second run (run ID 29) and export findings that are `Fixed`, enter this command:

   ```
   polyspace-access $login -export 125 -baseline 29  -resolution Fixed -output ./diff_fixed.txt

   Connecting to https://example-access-server:9443
   Connecting as jsmith
   Exporting results from RunId 125 and comparing to RunId 29
   Command Completed
   ```

   The command exports the list of findings that are fixed in the current run compared to the baseline run to file `diff_fixed.txt`.

You cannot specify more than one resolution type when you execute the command. To compare project runs for multiple resolution types, run the `polyspace-access -export` command for each resolution type.

See Also

Related Examples

- "Address Results in Polyspace Access Through Bug Fixes or Justifications"
- "Filter Results"

Export Results from Polyspace Access Web Server

Open or Export Results from Polyspace Access

Polyspace Access offers a centralized database where you can store Polyspace analysis results for sharing with your team and collaborative reviews. After you upload analysis results to the database, you can:

- View the results in your web browser.
- Open the results from any Polyspace desktop interface that is configured for Polyspace Access
- Export a list of results to a tab-separated value (TSV) file for further processing, such as applying custom filters and pass/fail criteria.
- Download results by using the `polyspace-access` . You use these downloaded results to merge review information between Polyspace Access projects . See also:

  - "Import Review Information from Existing Polyspace Access Projects" on page 27-5

  You cannot open results that you download with `polysace-access` in any Polyspace interface.

The rest of this topic discusses how to open Polyspace Access results in a desktop interface and how to export results to a TSV file.

Open Polyspace Access Results in a Desktop Interface

Before you open Polyspace Access results in a desktop interface, you must configure the Polyspace desktop interface to communicate with Polyspace Access. See "Register Polyspace Desktop User Interface".

To open results stored in the Polyspace Access database, go to **Access** > **Open Result** in the desktop interface, and follow the prompts. If you get a login request, use your Polyspace Access login credentials.

You can also open the desktop interface from the Polyspace Access web interface. On the toolstrip, click **Open in Desktop**. The desktop interface opens and shows the analysis results currently displayed in the Polyspace Access web interface.

Note In Linux, the desktop interface must already be open before you can view analysis results currently open in Polyspace Access.

Once you open results in the Polyspace desktop interface, changes you make to the **Status**, **Severity**, or comments fields are reflected in Polyspace Access after you save those changes.

If you open a local copy of the results that you uploaded to Polyspace Access in the desktop interface, the review fields in the **Result Details** pane are read-only. You cannot edit the **Status**, **Severity**, or comments fields.

Export Polyspace Access Results to a TSV File

You can export Polyspace Access results to a tab-separated values (TSV) file only from the command line by using the `polyspace-access` binary. When you export results, you generate a TSV file that lists results with most of the same results attributes as the "Results List in Polyspace Access Web Interface" on page 26-17. Each listed result also includes a URL through which you can open the

result in Polyspace Access. To filter the list of results you export, see the `polyspace-access` export options.

For example, to export all coding rules with status `Unreviewed` from project **myProject** stored in the **public** folder on Polyspace Access, open a command prompt terminal and enter:

```
polyspace-access -host hostName -port port ^
-export public/myProject -coding-rules -review-status Unreviewed ^
-output coding_rules.tsv
```

The command prompts you for your Polyspace Access login credentials, and then outputs file `coding_rules.tsv`.

`hostName` and *`port`* correspond to the host name and port number you specify in the URL of the Polyspace Access interface, for example `https://hostName:port/metrics/index.html`. If you are unsure about which host name and port number to use, contact your Polyspace Access administrator. Depending on your configuration, you might also have to specify the `-protocol` option in the command. See "Configure and Start the Cluster Admin".

See Also

`polyspace-access`

Related Examples

- "Add Custom Status in Polyspace Access Project" on page 27-3
- "Import Review Information from Existing Polyspace Access Projects" on page 27-5

Generate Report and Variables List from Polyspace Access

Note To generate reports of results on Polyspace Access at the command line, you must have a Polyspace Bug Finder Server or Polyspace Code Prover Server installation.

Suppose that you want to generate a report and export the variables list from the results of a Code Prover analysis that you can view in a Polyspace Access project.

To connect to Polyspace Access, provide a host name and your login credentials including your encrypted password. To encrypt your password, use the `polyspace-access` command and enter your user name and password at the prompt.

```
polyspace-access -encrypt-password
login: jsmith
password:
CRYPTED_PASSWORD LAMMMEACDMKEFELKMNDCONEAPECEEKPL
Command Completed
```

Store the login and encrypted password in a credentials file and restrict read and write permission on this file. Open a text editor, copy these two lines in the editor, then save the file as `myCredentials.txt` for example.

```
 -login jsmith
 -encrypted-password LAMMMEACDMKEFELKMNDCONEAPECEEKPL
```

To restrict the file permissions, right-click the file and select the **Permissions** tab on Linux or the **Security** tab on Windows.

To select a project to summarize in Polyspace Access, specify the run ID of the project. To obtain a list of projects with their latest run IDs, use the command `polyspace-access` with option `-list-project`.

```
polyspace-access -host myAccessServer -credentials-file myCredentials.txt -list-project
Connecting to https://myAccessServer:9443
Connecting as jsmith
Get project list with the last Run Id
Restricted/Code_Prover_Example (Code Prover) RUN_ID 14
public/Bug_Finder_Example (Bug Finder) RUN_ID 24
public/CP/Code_Prover_Example (Polyspace Code Prover) RUN_ID 16
public/Polyspace (Code Prover) RUN_ID 28
Command Completed
```

For more information on this command, see `polyspace-access`.

Generate a `Developer` report for results with run ID `16` from the Polyspace Access instance with host name `myAccessServer`. The URL of this instance of Polyspace Access is `https://myAccessServer:9443`.

```
SET template_path=^
"C:\Program Files\Polyspace\R2019a\toolbox\polyspace\psrptgen\templates"

polyspace-report-generator -credentials-file myCredentials.txt ^
-template %template_path%\Developer.rpt ^
-host myAccessServer ^
-run-id 16 ^
-output-name myReport
```

The command creates report `myReport.docx` by using the template that you specify. The report is stored in folder `Polyspace-Doc` on the path from which you called the command.

Generate a tab-delimited text file that contains a list of global variables in your code for the specified analysis results.

```
polyspace-report-generator -credentials-file myCredentials.txt^
-generate-variable-access-file ^
-host myAccessServer ^
-run-id 16
```

The list of global variables `Variable_View.txt` is stored in the same folder as the generated report. For more information on the exported variables list, see “View Exported Variable List” on page 25-12.

Review Workflows Common to All Platforms

30

Hide Known or Acceptable Results Using Code Annotations

Annotate Code and Hide Known or Acceptable Results

If a Polyspace analysis of your code finds known or acceptable defects or coding rule violations, you can suppress the defects or violations in subsequent analyses. Add code annotations indicating that you have reviewed the issues and that you do not intend to fix them.

You can add annotations through menu items in the Polyspace user interface (or IDE plugins) or by typing them directly in your code. For the general workflow of adding annotations:

- In the Polyspace desktop user interface, see "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" on page 23-2.
- In the Polyspace Access web interface, see "Address Results in Polyspace Access Through Bug Fixes or Justifications" on page 27-2.
- In IDEs using Polyspace as You Code plugins or extensions, see "Review Polyspace as You Code Results in IDEs".

This topic shows the annotation syntax.

Note that you cannot hide the run-time errors detected with Code Prover from your source code even with code annotations. However, like all other results, the review information associated with a run-time error is extracted from the corresponding code annotation and shown with the result.

Code Annotation Syntax

To add comments directly to your source file, use the Polyspace annotation syntax. The syntax is not case sensitive, and has the following format. Both C style comments within `/* */` and C++ style comments starting with `//` are supported.

Annotating Single Line of Code

To annotate a result on the current line of code (including macros), use this syntax:

```
line of code; /* polyspace Family:Result_name */
```

For instance:

```
var++; /* polyspace DEFECT:INT_OVFL */
```

Annotations begin with the keyword `polyspace` and must include the *`Family`* and *`Result_name`* field values.

You can optionally specify a *`Status`*, *`Severity`*, and *`Comment`* field value:

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

For instance:

```
var++; /* polyspace DEFECT:INT_OVFL [Justified:Low] "Overflow taken into 
                                         account."*/
```

If you do not specify a status, Polyspace considers the result justified, and assigns the status `No action planned` to the result.

For further details, see "Annotation Syntax Details" and "Syntax Examples".

Annotating Code Block

To annotate a block of code, use the following syntax. Note that the annotations apply only to the block of code itself and not to bodies of functions called from the block.

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

  ```
  line of code; /* polyspace +n Family:Result_name */
  ```
- Annotation for block of code:

  ```
  /* polyspace-begin Family:Result_name */
  {
     block of code
  }
  /* polyspace-end Family:Result_name */
  ```

Optionally, specify a status, severity and comment.

If annotations for results with the same *Family* and *Result_name* are nested, the innermost annotation is used.

For example, in this code, the annotation on line 9 is applied instead of the block annotation, but the block annotation is applied to the violation on line 7.

```
1  /*polyspace-begin MISRA-C:14.9 [To fix:High] */
2  int main(void)
3  {
4      int x = 1;
5      int y = x / 2;
6
7      if (y < 0) /* Block annotation is applied to this violation of MISRA-C:14.9*/
8          y++;
9      if (x > y) /*polyspace MISRA-C:14.9 [Justified:Low] */
10         return x;
11     return x;
12 }
13 /*polyspace-end MISRA-C:14.9 [To fix:High] "Block annotation"*/
```

If you apply an annotation to multiple lines of code, the annotation does not apply to green checks in the code. When you rerun the analysis these green checks are not considered justified, and their *Status* and *Severity* in the **Results List** do not change to the *Status* and *Severity* of the annotation.

When you annotate a code block, the annotation applies only to the issues that arise from within the block. For instance, say you have a function call in the annotated block, and the body of the function gives rise to a violation. This violation is not affected by the annotation around the code block where the function is called.

For further details, see “Annotation Syntax Details” and “Syntax Examples”.

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” and “Syntax Examples”.

Annotation Syntax Details

To replace the different annotation fields with their allowed values, use the values in this table or see the examples.

Field	Allowed Value
Family	Type of analysis result: • DEFECT (Polyspace Bug Finder) • RTE, for run-time checks (Polyspace Code Prover) • CODE-METRICS, for function-level code complexity metrics • VARIABLE, for global variables (Polyspace Code Prover) • MISRA-C or MISRA2004 for MISRA C: 2004 rule violations. These annotations also apply to MISRA C: 2012 violations based on the mapping between the two standards. The mapping allows you to reuse your justifications for the older standard when migrating to the newer one. See “Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results” on page 20-5. • MISRA-AC-AGC for violations of MISRA C:2004 rules applicable to generated code • MISRA-C3 or MISRA2012 for MISRA C: 2012 rule violations. The annotation works even for the rules applicable to generated code. • CERT-C for CERT C coding standard violations • CERT-CPP for CERT C++ coding standard violations • ISO-17961 for ISO/IEC TS 17961 coding standard violations • MISRA-CPP for MISRA C++ rule violations • AUTOSAR-CPP14 for AUTOSAR C++14 rule violations • JSF for JSF++ rule violations • GUIDELINE for software complexity guidelines. • CUSTOM for violations of custom coding rules To specify all analysis results, use the asterisk character *:*. See “Syntax Examples”.

Field	Allowed Value
`Result_name`	For `DEFECT`, use short names of checkers. See “Short Names of Bug Finder Defect Groups and Defect Checkers”. For `RTE`, use short names of run-time checks. See “Short Names of Code Prover Run-Time Checks” on page 30-12. For `CODE-METRICS`, use short names of code complexity metrics. See “Short Names of Code Complexity Metrics” on page 30-14. Note that project and file metrics cannot be justified using code annotations. For `VARIABLE`, the only allowed value is the asterisk character " * ". For coding standard violations, specify the rule number or numbers. For software complexity guidelines, use acronyms for the guidelines. See pages for individual guidelines in “Guidelines”. To specify all parts of a result name `[MISRA2012:17.*]` or all result names in a family `[DEFECT:*]`, use the asterisk character. See “Syntax Examples”.
`Status`	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: • `Unreviewed` • `To investigate` • `To fix` • `Justified` • `No action planned` • `Not a defect` • `Other` 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.

Field	Allowed Value
`Severity`	Text to indicate how critical you consider the error in your code. This value populates the **Severity** column in the **Results List** pane as: • `Unset` • `High` • `Medium` • `Low` 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`.
`Comment`	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. 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.

Syntax Examples

Suppress a Single Defect

Enter an annotation on the same line as the defect and specify the *`Family`* (`DEFECT`) and the *`Result_name`* (`INT_OVFL`). When you do not specify a status, Polyspace assigns the status `No action planned`, and then suppresses the result in subsequent analyses.

```
int var = INT_MAX;
var++; /* polyspace DEFECT:INT_OVFL */
```

Suppress a Single Coding Standard Violation

Justify a coding standard violation, for instance, a CERT-C violation.

Enter an annotation on the same line as the violation and specify the *`Family`* (`CERT-C`) and the *`Result_name`* (the rule number, for instance, `STR31-C`). Assign the status `Justified`, severity `Low` and a comment.

```
line of code; /* polyspace CERT-C:STR31-C [Justified:Low] "Overflow cannot happen
                                            because of external constraints." */
```

Suppress All MISRA C: 2012 Violations Over Multiple Lines

Enter an annotation with +n between `polyspace` and the *`Family`*:*`Result_name`* entries. The annotation applies to the same line and the n following lines.

This annotation applies to lines 4–7. The line count includes code, comments, and blank lines.

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

Suppress All Code Metrics on Function

To annotate function-level code complexity metrics, in the function definition, enter an annotation on the same line as the function name.

This annotation suppresses all code complexity metrics for function `func`:

```
char func(char param) { //polyspace CODE-METRICS:*
   ...
}
```

Specify Multiple Families in the Same Annotation

Enter each family separated by a space. This annotation applies to all MISRA C:2012 rules 17 and to all run-time checks.

```
line of code; /* polyspace MISRA2012:17.* RTE:* */
```

Specify Multiple Result Names in the Same Annotation

After you specify the *`Family`* (`DEFECT`), enter each *`Result_name`* separated by a comma.

```
system("rm ~/.config"); /* polyspace DEFECT:UNSAFE_SYSTEM_CALL,RETURN_NOT_CHECKED */
```

Suppress Result Showing Global Variable Usage

To justify a Code Prover result showing global variable usage, for instance, an unused global variable, enter the annotation next to the variable declaration.

For instance, to suppress a global variable result with a `Justified` status, `Low` severity and some comments, you can enter an annotation like this:

```
int var; /* polyspace VARIABLE:* [Justified: Low] "Storage repo for later use"*/
```

Add Explanatory Notes to Annotation

After you specify a *`Family`* and a *`Result_name`*, you can add a *`Comment`* with additional information for your justification. You can provide a comment for all families and result names, or a comment for each family or result name.

```
//Single comment

line of code; /* polyspace DEFECT:BAD_FREE MISRA2004:* "OK Defect and MISRA" */
//Multiple comments incorrect syntax:

line of code; /* polyspace DEFECT:* "OK defect" MISRA2004:5.2 "OK MISRA" */

//Multiple comments correct syntax:
line of code; /* polyspace DEFECT:* "OK defect" polyspace MISRA2004:5.2 "OK MISRA" */
```

In annotations, Polyspace ignores all text following double quotes. To specify additional `Family:Result_name, [Status:Severity]` or `Comment` entries, you must reenter the keyword `polyspace` after text in double quotes.

Set Status and Severity

You can specify allowed values on page 30-2 or enter custom values for status and severity. A custom severity entry is appended to the status and stored as a custom **Status** in the user interface.

```
//Set Status only
line of code; /* polyspace DEFECT:* [To fix] "some comment" */

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

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

Justify Violations in a Code Block

Use annotation to justify violations arising from a block of code. For instance, consider this code:

```
double foo(void){
    constexpr int speedLimit = 65;
    constexpr double coeff = 0.2;
    int flag{0};
    int negOne{-1};
    //...
    return (flag)?speedLimit*coeff*negOne
    : speedLimit*coeff*negOne - 35; //Noncompliant
}

int main(){

    /* polyspace-begin AUTOSAR-CPP14:A5-1-1 [Justified: Low]"Known Constant"*/
    //....
    for(int i = 0; i<10;++i){
        foo();
        //...
    }

    /* polyspace-end AUTOSAR-CPP14:A5-1-1  [Justified: Low]"Known Constant"*/
    return 1;
}
```

The `for` loop has hard coded literal as the loop boundary, which is a violation of `AUTOSAR C++14 Rule A5-1-1`. By annotating the block with the syntax `/*polyspace begin...*/ {...}/*polyspace-end...*/`, the violation in the line `for(int i = 0; i<10;++i){` appears as a justified defect in the results list.

The annotated block contains a call to the function `foo()`. The annotations do not apply to the body of this function. For instance, the violation in the return statement of `foo()` appears as an unreviewed defect.

Code Annotation Warnings

If you enter a code annotation incorrectly or the annotation no longer applies, the analysis log contains a warning:

```
Warning: These Polyspace annotations do not apply to the current code
```

The warning can mean one of the following:

- The issue is no longer detected because of code fixes or changes in analysis configuration.

  For instance, an annotation such as:

  ```
  /* polyspace RTE:IDP [No action planned:Low] */
  ```

  Might no longer apply because the **Illegally dereferenced pointer** check (annotated as `IDP`), which was previously red or orange, is now green.
- The annotation syntax is incorrect.

  An annotation beginning with `polyspace` followed by a word and then a `:` (colon) such as:

  ```
  // polyspace Family :
  ```

  is considered as a Polyspace annotation justifying a result. If the word *`Family`* following `polyspace` is not a type of Polyspace result such as `DEFECT` or `RTE`, the analysis considers the annotation as invalid and shows the warning. For instance, this annotation triggers the warning:

  ```
  // polyspace TODO: Fix in March dev cycle
  ```

  since `TODO` is not a type of Polyspace result. To avoid these warnings, use another separator, for instance, instead of a colon. For the full list of result types, see “Code Annotation Syntax” on page 30-2.

Ignoring Code Annotations

In some cases, you might want to run a clean analysis as if the results have not been previously reviewed. For instance, you might want to perform a worst-case analysis where you see all previously justified results.

You can use the option `-ignore-code-annotations` to run such an analysis with no history. The analysis ignores the code annotations and shows all annotated results without any review information taken from the annotations.

See also `-ignore-code-annotations`.

See Also

`-xml-annotations-description` | `-ignore-code-annotations`

More About

- "Define Custom Annotation Format" on page 30-20
- "Short Names of Code Prover Run-Time Checks" on page 30-12
- "Short Names of Code Complexity Metrics" on page 30-14

Short Names of Code Prover Run-Time Checks

When annotating your code to justify checks or creating custom software quality objectives, you use short names of Code Prover run-time checks instead of the full names. The following table lists the short names for individual run-time checks.

Check	Acronym
Absolute address usage	ABS_ADDR
AUTOSAR runnable not implemented	AUTOSAR_NOIMPL
Correctness condition	COR
Division by zero	ZDV
Function not called	FNC
Function not reachable	FNR
Function not returning value	FRV
Illegally dereferenced pointer	IDP
Incorrect object oriented programming	OOP
Invalid C++ specific operations	CPP
Invalid operation on floats	INVALID_FLOAT_OP
Invalid result of AUTOSAR runnable implementation	AUTOSAR_IMPL
Invalid shift operations	SHF
Invalid use of AUTOSAR runtime environment function	AUTOSAR_USE
Invalid use of standard library routine	STD_LIB
Non-initialized local variable	NIVL
Non-initialized pointer	NIP
Non-initialized variable	NIV
Non-terminating call	NTC
Non-terminating loop	NTL
Null this-pointer calling method	NNT
Out of bounds array index	OBAI
Overflow	OVFL
Return value not initialized	IRV
Subnormal float	SUBNORMAL
Uncaught exception	EXC
Unreachable code	UNR
User assertion	ASRT

See Also

More About

- “Annotate Code and Hide Known or Acceptable Results”

Short Names of Code Complexity Metrics

When annotating your code to justify metrics or creating custom software quality objectives, you use short names of code complexity metrics instead of the full names. The following table lists the short names for code complexity metrics.

Note that you can only annotate your code for function level code complexity metrics only.

Project Metrics

Code Metric	Acronym
Number of Direct Recursions	AP_CG_DIRECT_CYCLE
Number of Header Files	INCLUDES
Number of Files	FILES
Number of Recursions	AP_CG_CYCLE
Program Maximum Stack Usage	PROG_MAX_STACK
Program Minimum Stack Usage	PROG_MIN_STACK

File Metrics

Code Metric	Acronym
Comment Density	COMF
Estimated Function Coupling	FCO
Number of Lines	TOTAL_LINES
Number of Lines Without Comment	LINES_WITHOUT_CMT

Function Metrics

Code Metric	Acronym
Cyclomatic Complexity	VG
Higher Estimate of Size of Local Variables	LOCAL_VARS_MAX
Language Scope	VOCF
Lower Estimate of Size of Local Variables	LOCAL_VARS_MIN
Minimum Stack Usage	MIN_STACK
Maximum Stack Usage	MAX_STACK
Number of Call Levels	LEVEL
Number of Call Occurrences	NCALLS
Number of Called Functions	CALLS
Number of Calling Functions	CALLING

Code Metric	Acronym
Number of Executable Lines	FXLN
Number of Function Parameters	PARAM
Number of Goto Statements	GOTO
Number of Instructions	STMT
Number of Lines Within Body	FLIN
Number of Local Non-Static Variables	LOCAL_VARS
Number of Local Static Variables	LOCAL_STATIC_VARS
Number of Paths	PATH
Number of Return Statements	RETURN

See Also

More About

- "Annotate Code and Hide Known or Acceptable Results" on page 30-2

Annotate Code for Known or Acceptable Results (Not Recommended)

Note Starting R2017b, Polyspace uses a simpler annotation format. See “Annotate Code and Hide Known or Acceptable Results” on page 30-2.

If Polyspace finds defects in your code that you cannot or will not fix, you can add annotations to your code. These annotations are code comments that indicate known or acceptable defects or coding rule violations. By using these annotations, you can:

- Avoid rereviewing defects or coding rule violations from previous analyses.
- Preserve review comments and classifications.

Note Source code annotations do not apply to code comments. You cannot annotate these rules:

- MISRA C:2004 Rules 2.2 and 2.3
- MISRA C:2012 Rules 3.1 and 3.2
- MISRA-C++ Rule 2-7-1
- JSF++ Rules 127 and 133

Add Annotations from the Polyspace Interface

This method shows you how to convert review comments and classifications in the Polyspace interface into code annotations.

1. On the **Results List** or **Result Details** pane, assign a **Severity**, **Status**, and **Comment** to a result.
   a. Click a result.
   b. From the **Severity** and **Status** dropdown lists, select an option.
   c. In the **Comment** field, enter a comment or keyword that helps you easily recognize the result.
2. On the **Results List** pane, right-click the commented result and select **Add Pre-Justification to Clipboard**. The software copies the severity, status, and comment to the clipboard.
3. Right-click the result again and select **Open Editor**. The software opens the source file to the location of the defect.
4. Paste the contents of your clipboard on the line immediately before the line containing the defect or coding rule violation.

   You can see your review comments as a code comment in the Polyspace annotation syntax, which Polyspace uses to repopulate review comments on your next analysis.
5. Save your source file and rerun the analysis.

   On the **Results List** pane, the software populates the **Severity**, **Status**, and **Comment** columns for the defect or rule violation that you annotated. These fields are read only because they are

populated from your code annotation. If you use a specific keyword or status for your annotations, you can filter your results to hide or show your annotated results. For more information on filtering, see “Filter and Group Results in Polyspace Desktop User Interface” on page 24-2.

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”.

Word	Allowed Values
Type	The type of results: • `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
Kind1,[Kind2],...	For defects, run-time checks and code metrics, use the short names of checkers. See: • “Short Names of Bug Finder Defect Groups and Defect Checkers” • “Short Names of Code Prover Run-Time Checks” on page 30-12 For coding rule violations, specify the rule number or numbers. For global variables, the only allowed value is `ALL`. If you want the comment to apply to all possible defects or coding rules, specify `ALL`.
Severity	Text that indicates how critical you consider the defect. Enter one of the following: • `Unset` • `High` • `Medium` • `Low` This text populates the **Severity** column on the **Results List** pane.
Status	Text that indicates how you intend to correct the error in your code. Enter one of the following or any other text: • `Unreviewed` • `To investigate` • `To fix` • `Justified` • `No action planned` • `Not a defect` • `Other` This text populates the **Status** column on the **Results List** pane. The status is also used in Polyspace Access to determine whether a result is justified. To justify a result, use `Justified`, `No action planned` or `Not a defect`.
Notes	Additional comments, such as a keyword or an explanation for the status and severity.

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”.

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="RTE"
    Result_Name="ZDV"/>
<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	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.
			
			`//myKeyword 50, 51 Block_on`
			`Code block 1;`
			`...`

		END	Stops the application of a rule identifier declared by a corresponding expression with attribute `Mode="BEGIN"`.
			
			`//myKeyword 50, 51 Block_on`
			`Code block 1;`
			`...`
			`More code;`
			`//myKeyword 50 Block_off`
			
			Only rule identifier 50 is turned off. Rule identifier 51 still applies.

		END_ALL	Stops all rule identifiers declared by an expression with attribute `Mode="BEGIN"`.
			
			`//myKeyword 50, 51 Block_on`
			`Code block 1;`
			`...`
			`More code;`
			`//myKeyword Block_off_all`
			
			Rule identifiers 50 and 51 are turned off.

Regex	Required	Regular expression search string	See “Regular Expressions”. `Regex="myKeyword\s+(\w+(\s*,\s*\w+)*)"` matches these expressions:
			
			`// myKeyword 50, 51`
			`/* myKeyword ALL_MISRA, 100 */`

Attribute	Use	Value	Example
Rule_Identifier_Position	Required, except when you set Mode="END_ALL"	Integer	The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression. <Expression Mode="GOTO_INCREMENT" Regex="myKeyword\s+(\+\d+\s) (\w+(\s*,\s*\w+)*)" Increment_Position="1" Rule_Identifier_Position="2"/> **Note** Enter the regex expression on a single line when you edit your XML file. The search expression for the rule identifier \w+(\s*,\s*\w+)* is after the second opening parenthesis of the regular 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. <Expression Mode="GOTO_INCREMENT" Regex="myKeyword\s+(\+\d+\s) (\w+(\s*,\s*\w+)*)" Increment_Position="1" Rule_Identifier_Position="2"/> **Note** Enter the regex expression on a single line when you edit your XML file. The search expression for the increment \+\d+\s is after the first opening parenthesis of the regular expression.
Status_Position	Optional	Integer	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.
Severity_Position	Optional	Integer	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.

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 `Justified by annotation in source:`
`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 `Case_Insensitive="true"`, these annotations are equivalent: `//MYKEYWORD ALL_MISRA BLOCK_ON` `//mykeyword all_misra block_on`

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 `annotations_description.xml`
`Family`	Required	Corresponds to Polyspace results family. For a list of allowed values, see allowed values on page 30-12.	See the mapping section of `annotations_description.xml`
`Result_Name`	Required	Corresponds to Polyspace result names. For a list of allowed values, see allowed values on page 30-12.	See the mapping section of `annotations_description.xml`

Define Multiple Custom Annotation Syntaxes

To define more than one annotation syntax, in your XML file, specify a comma separated list of keywords associated with each syntax for the `Search_For_Keywords` attribute.

For example, if you use custom annotations that follow these patterns to annotate violations of MISRA C: 2012 rules:

```
int func(int p) //customSyntax M123 $ customSyntax M124
{
    int i;
    int j = 1;

    i = 1024 / (j - p);
    return i;
}

int func2(void){ //otherCustomSyntax 50
    int x=func(2);
    return x;
}
```

Enter the following in the XML file where you define the custom annotation syntax.

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

<Annotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:noNamespaceSchemaLocation="annotations_xml_schema.xsd"
             Group="multipleCustomSyntax">
    <!-- Enter comma separated list of keywords -->
  <Expressions Search_For_Keywords="customSyntax,otherCustomSyntax"
              Separator_Result_Name="$" >

    <!-- This section defines the annotation syntax format -->
   <Expression Mode="SAME_LINE"
               Regex="customSyntax\s(\w+(\s*,\s*\w+)*)"
               Rule_Identifier_Position="1"
               />
    <Expression Mode="SAME_LINE"
               Regex="otherCustomSyntax\s(\w+(\s*,\s*\w+)*)"
               Rule_Identifier_Position="1"
               />
 </Expressions>
  <!-- This section maps the user annotation to the Polyspace
  annotation syntax -->
 <Mapping>
    <!-- Mapping for customSyntax rules -->
    <Result_Name_Mapping  Rule_Identifier="M123" Family="MISRA-C3" Result_Name="8.7"/>
       <Result_Name_Mapping  Rule_Identifier="M124" Family="MISRA-C3" Result_Name="D4.6"/>
    <!-- Mapping for otherCustomSyntax rules  -->
    <Result_Name_Mapping  Rule_Identifier="50" Family="MISRA-C3" Result_Name="8.4"/>
 </Mapping>
</Annotations>
```

When you use multiple custom annotations, each rule identifier must be unique. For instance, in the preceding example, you cannot reuse rule identifier `M123` with `otherCustomSyntax`.

See Also

`-xml-annotations-description`

More About

- “Annotation Description Full XML Template” on page 30-28
- “Annotate Code and Hide Known or Acceptable Results” on page 30-2

- “Fix Errors Applying Custom Annotation Format for Polyspace Results” on page 34-97

Annotation Description Full XML Template

This table lists all the elements, attributes, and values of the XML that you can use to define an annotation format and map it to the Polyspace annotation syntax. For an example of how to edit an XML to define annotation syntax, see “Define Custom Annotation Format” on page 30-20.

Element	Attribute	Use	Value
`Annotations`	`Group`	Required	User defined string. For example, "Custom Annotations"
`Expressions`	`Search_For_Keywords`	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”.
	`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`
			`END`

Element	Attribute	Use	Value
			END_ALL
			NEXT_CODE_LINE The annotation applies to the next line of code. Comments and blank lines are ignored.
			GOTO_LABEL
			LABEL
			XML_START
			XML_CONTENT The annotation for this expression must be on a single line.
			XML_END
	Regex	Required	Regular expression search string that matches the pattern of your annotation.
	Rule_Identifier_Position	Required, except when you set Mode="END_ALL" or "LABEL"	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Increment_Position	Required only when you set Mode="GOTO_INCREMENT"	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Status_Position	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.

Element	Attribute	Use	Value
	Severity_Position	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Comment_Position	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Label_Position	Required only when you set Mode="GOTO_LABEL" or "LABEL"	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Case_Insensitive	Optional	True or false. When you do not declare this attribute, the default value is false.
	Is_Pragma	Optional	True or false. When you do not declare this attribute, the default value is false. Set this attribute to true if you want to declare your annotation using a pragma instead of a comment.
	Applies_Also_On_Same_Line	Optional	True or false. When you do not declare this attribute, the default value is true. Use this attribute to enable annotations with the old Polyspace syntax to apply on the same line of code.

Element	Attribute	Use	Value
Mapping	None	None	None
Result_Name_Mapping	Rule_Identifier	Required	User defined
	Family	Required	Corresponds to Polyspace results family. For a list of allowed values, see allowed values on page 30-12.
	Result_Name	Required	Corresponds to Polyspace result names. For a list of allowed values, see allowed values on page 30-12.

Example

This example code covers some of the less commonly used attributes for defining annotations in XML.

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


<Annotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:noNamespaceSchemaLocation="annotations_xml_schema.xsd"
             Group="XML Template">

  <Expressions Separator_Result_Name=","
               Search_For_Keywords="myKeyword">

    <Expression Mode="GOTO_LABEL"
                Regex="(\A|\W)myKeyword\s+S\s+(\d+(\s*,\s*\d+)*)\s+([a-zA-Z_-]\w+)"
                Rule_Identifier_Position="2"
                Label_Position="4"

                />

     <Expression Mode="LABEL"
                Regex="(\A|\W)myKeyword\s+L:(\w+)"
                Label_Position="2"

                />
     <!-- Annotation applies starting current line until
        next declaration of label word "myLabel"
         Example:

         code; // myKeyword S 100 myLabel
         ...
         more code;
         // myKeyword L myLabel
    -->

    <Expression Mode="BEGIN"
                Regex="#\s*pragma\s+myKeyword_MESSAGES_ON\s+(\w+)"
                Rule_Identifier_Position="1"
                Is_Pragma="true"
                />
    <!-- Annotation declared with pragma instead of comment
        Example:#pragma myKeyword_MESSAGES_ON 100 -->


    <!-- Comment declaration with XML format-->

    <!-- XML_START must be declared before XML_CONTENT -->
    <Expression Mode="XML_START"
                Regex="<\s*myKeyword_COMMENT\s*>"

                />
    <!-- Example: <myKeyword_COMMENT>  -->

    <Expression Mode="XML_CONTENT"
                Regex="<\s*(\d*)\s*>(((?![*]/)(?!<).)*)</\s*(\d*)\s*>"
                Rule_Identifier_Position="1"
                Comment_Position="2"

                />
    <!-- Example: <100>This is my comment</100>
       XML_CONTENT must be declare on a single line.

       <100>This is my comment
       </100>
       is incorrect.
       -->

    <Expression Mode="XML_END"
                Regex="</\s*myKeyword_COMMENT\s*>"

                />
    <!--    Example: </myKeyword_COMMENT> -->
  </Expressions>

  <Mapping>

    <Result_Name_Mapping Rule_Identifier="100" Family="MISRA-C" Result_Name="4.1"/>
      </Mapping>
</Annotations>
```

See Also

`-xml-annotations-description`

More About

- "Annotate Code and Hide Known or Acceptable Results" on page 30-2

Advanced Review Workflows

Evaluate Polyspace Code Prover Results Against Software Quality Objectives

Instead of evaluating all results of a Code Prover analysis, you can first define a set of criteria that the analyzed project must meet and compare the Code Prover results against those criteria. The Software Quality Objectives or SQOs are a set of thresholds against which you can compare your verification results. You can develop a review process based on the Software Quality Objectives. In your review process, you consider only those results that cause your project to fail a certain SQO level.

You can use a predefined SQO levels or define your own. To customize SQO levels, see "Customize Software Quality Objectives" on page 26-14.

Specifications of SQO Levels

Following are the quality thresholds specified by each predefined SQO.

SQO Level 1

Metric	Threshold Value
Comment density of a file	20
Number of paths through a function	80
Number of `goto` 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 `return` statements in a function	1
Language scope, an indicator of the cost of maintaining or changing functions. Calculated as follows: *(N1+N2) / (n1+n2)* • *n1* — Number of different operators • *N1* — Total number of operators • *n2* — Number of different operands • *N2* — Total number of operands	4
Number of recursions	0
Number of direct recursions	0

Metric	Threshold Value
Number of unjustified violations of the following MISRA C:2004 rules: • 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: • 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: • 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

SQO Level 2

In addition to all the requirements of SQO Level 1, this level includes the following thresholds:

Metric	Threshold Value
Number of unjustified red checks	0
Number of unjustified `Non-terminating call` and `Non-terminating loop` checks	0

SQO Level 3

In addition to all the requirements of SQO Level 2, this level includes the following thresholds:

Metric	Threshold Value
Number of unjustified gray `Unreachable code` checks	0

SQO Level 4

In addition to all the requirements of SQO Level 3, this level includes the following thresholds:

Metric	Threshold Value
Percentage of justified orange checks, calculated as the number of green and justified orange checks divided by the total number of green and orange checks.	`Invalid C++ specific operations`: 50
	`Correctness condition`: 60
	`Division by zero`: 80
	`Uncaught exception`: 50
	`Function not returning value`: 80
	`Illegally dereferenced pointer`: 60
	`Return value not initialized`: 80
	`Non-initialized local variable`: 80
	`Non-initialized pointer`: 60
	`Non-initialized variable`: 60
	`Null this-pointer calling method`: 50
	`Incorrect object oriented programming`: 50
	`Out of bounds array index`: 80
	`Overflow`: 60
	`Invalid shift operations`: 80
	`User assertion`: 60

SQO Level 5

In addition to all the requirements of SQO Level 4, this level includes the following thresholds:

Metric	Threshold Value
Number of unjustified violations of the following MISRA C:2004 rules: • 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: • 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
Number of unjustified violations of the following MISRA C++ rules: • 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
Percentage of justified orange checks, calculated as the number of green and justified orange	`Invalid C++ specific operations`: 70
	`Correctness condition`: 80

Metric	Threshold Value
checks divided by the total number of green and orange checks.	`Division by zero`: 90
	`Uncaught exception`: 70
	`Function not returning value`: 90
	`Illegally dereferenced pointer`: 70
	`Return value not initialized`: 90
	`Non-initialized local variable`: 90
	`Non-initialized pointer`: 70
	`Non-initialized variable`: 70
	`Null this-pointer calling method`: 70
	`Incorrect object oriented programming`: 70
	`Out of bounds array index`: 90
	`Overflow`: 80
	`Invalid shift operations`: 90
	`User assertion`: 80

SQO Level 6

In addition to all the requirements of SQO Level 5, this level includes the following thresholds:

Metric	Threshold Value
Percentage of justified orange checks, calculated as the number of green and justified orange checks divided by the total number of green and orange checks.	`Invalid C++ specific operations`: 90
	`Correctness condition`: 100
	`Division by zero`: 100
	`Uncaught exception`: 90
	`Function not returning value`: 100
	`Illegally dereferenced pointer`: 80
	`Return value not initialized`: 100
	`Non-initialized local variable`: 100
	`Non-initialized pointer`: 80
	`Non-initialized variable`: 80
	`Null this-pointer calling method`: 90
	`Incorrect object oriented programming`: 90
	`Out of bounds array index`: 100
	`Overflow`: 100
	`Invalid shift operations`: 100
	`User assertion`: 100

Exhaustive

In addition to all the requirements of SQO Level 6, this level includes the following thresholds. The thresholds for coding rule violations apply only if you check for coding rule violations.

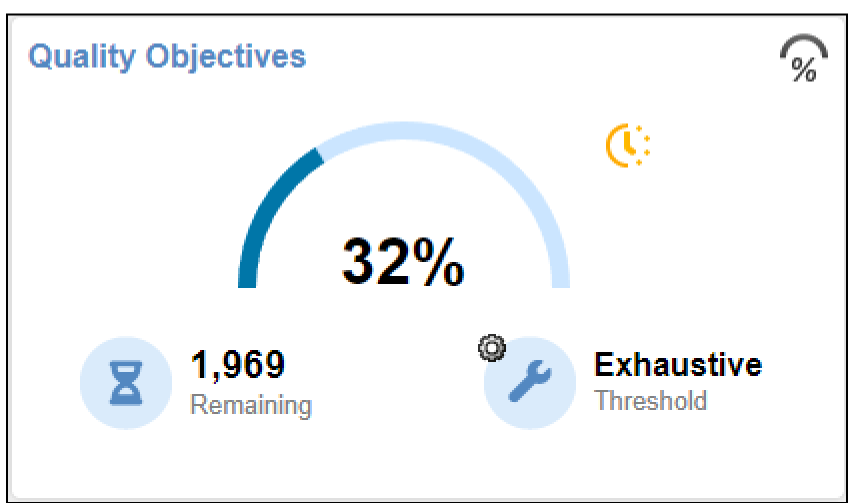
Metric	Threshold Value
Number of unjustified MISRA C and MISRA C++ coding rule violations	0
Number of unjustified red checks	0
Number of unjustified `Non-terminating call` and `Non-terminating loop` checks	0
Number of unjustified gray `Unreachable code` checks	0
Percentage of justified orange checks, calculated as the number of green and justified orange checks divided by the total number of green and orange checks.	100

For information on the rationales behind these levels, see Software Quality Objectives for Source Code.

Compare Verification Results Against Software Quality Objectives

You can compare your verification results against SQOs either in the Polyspace Access web interface or the Polyspace desktop user interface.

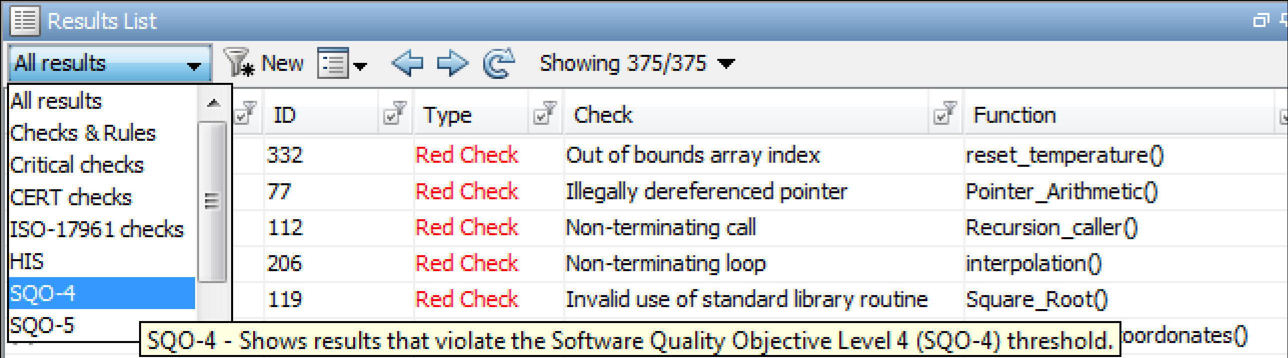
- In the Polyspace Access web interface, you can first determine whether your project fails to attain a certain Quality Objective threshold by looking at the **Quality Objectives** card on the **Project Overview** dashboard.



The card shows the percentage of results that you have already fixed or justified in order to attain the threshold. Click the number of remaining findings to open those findings in the **Results List**. For a more detailed view of the quality of your code against all quality objectives thresholds, open the **Quality Objectives** dashboard. For more information, see the “Quality Objectives Dashboard in Polyspace Access” on page 26-12.

You can also generated reports that show the **PASS** or **FAIL** status using the templates `SoftwareQualityObjectives_Summary` and `SoftwareQualityObjectives`. See `Bug Finder and Code Prover report (-report-template)`.

- In the Polyspace user interface, you can use the menu in the **Results List** toolbar to display only those results that you must fix or justify to attain a certain Software Quality Objective.



To activate the SQO options in this menu, select **Tools** > **Preferences**. On the **Review Scope** tab, select **Include Quality Objectives Scope**.

Note You cannot use the menu in the user interface to suppress red or gray checks. Therefore, you cannot directly compare your project against predefined SQO levels 1, 2 and 3 in the Polyspace user interface. However, in the Polyspace Access web interface, you can compare your project against all predefined SQO levels.

Customize SQO Levels

To customize SQOs:

- In the Polyspace Access web interface, see “Customize Software Quality Objectives” on page 26-14.
- In the Polyspace desktop user interface, review scopes are used to implement SQOs. See how to create your own review scopes in one of these topics:
  - “Limit Display of Orange Checks in Polyspace Desktop User Interface” on page 33-14
  - “Compute Code Complexity Metrics Using Polyspace” on page 16-47

See Also

Related Examples

- “Filter and Sort Results in Polyspace Access Web Interface” on page 28-8
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 27-2

Justify Coding Rule Violations Using Code Prover Checks

Coding rules are good practices that you observe for safe and secure code. Using the Polyspace coding rule checkers, you find instances in your code that violate a coding rule standard such as MISRA. If you run Code Prover, you also see results of checks that find run-time errors or prove their absence. In some cases, the two kinds of results can be used together for efficient review. For instance, you can use a green Code Prover check as rationale for not fixing a coding rule violation (justification).

If you run MISRA checking in Code Prover, some of the checkers use Code Prover static analysis under the hood to find MISRA violations. The MISRA checker in Code Prover is more rigorous compared to Bug Finder because Code Prover keeps precise track of the data and control flow in your code. For instance:

- `MISRA C:2012 Rule 9.1`: The rule states that the value of an object with automatic storage duration shall not be read before it has been set. Code Prover uses the results of a `Non-initialized local variable` check to determine the rule violations.
- MISRA C:2004 Rule 13.7: The rule states that the Boolean operations whose results are invariant shall not be permitted. Code Prover uses the results of an `Unreachable code` check to identify conditions that are always true or false.

In some other cases, the MISRA checkers do not suppress rule violations even though corresponding green checks indicate that the violations have no consequence. You have the choice to do one of these:

- Strictly conform to the standard and fix the rule violations.
- Manually justify the rule violations using the green checks as rationale.

  Set a status such as `No action planned` to the result and enter the green check as rationale in the result comments. You can later filter justified results using that status.

The following sections show examples of situations where you can justify MISRA violations using green Code Prover checks.

Rules About Data Type Conversions

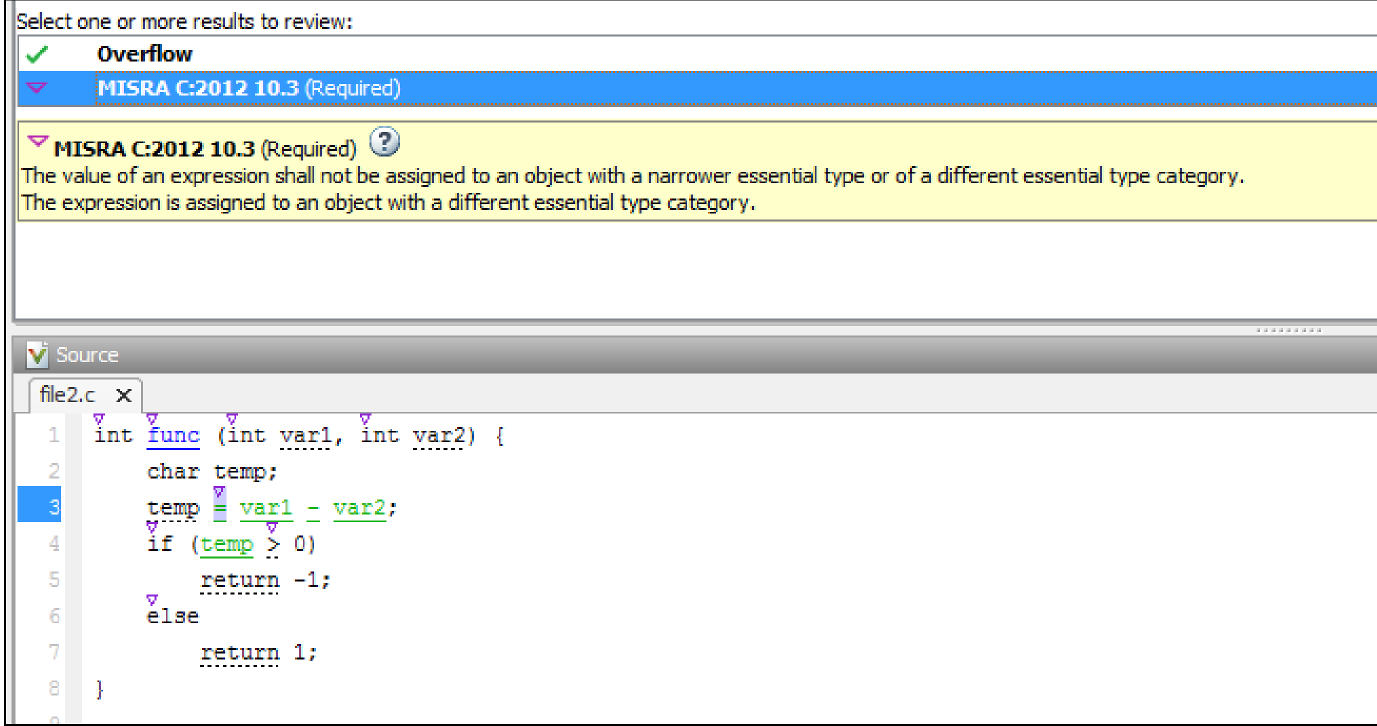
In some cases, implicit data type conversions are okay if the conversion does not cause an overflow.

In the following example, the line `temp = var1 - var2;` violates `MISRA C:2012 Rule 10.3`. The rule states that the value of an expression shall not be assigned to an object of a different essential type category. Here, the difference between two `int` variables is assigned to a `char` variable. You can justify this particular rule violation by using the results of a Code Prover `Overflow` check.

```
int func (int var1, int var2) {
    char temp;
    temp = var1 - var2;
    if (temp > 0)
        return -1;
    else
        return 1;
}

double read_meter1(void);
double read_meter2(void);

int main(char arg, char* argv[]) {
    int meter1 = (read_meter1()) * 10;
    int meter2 = (read_meter2()) * 999;
    int tol = 10;
    if((meter1 - meter2)> -tol && (meter1 - meter2) < tol)
        func(meter1, meter2);
    return 0;
}
```

Consider the rationale behind this rule. The use of implicit conversions between types can lead to unintended results, including possible loss of value, sign, or precision. For a conversion from `int` to `char`, a loss of sign or precision cannot happen. The only issue is a potential loss of value if the difference between the two `int` variables overflows.

Run Code Prover on this code. On the **Source** pane, click the = in `temp = var1 - var2;`. You see the expected violation of MISRA C:2012 Rule 10.3, but also a green **Overflow** check.



The green check indicates that the conversion from `int` to `char` does not overflow.

MISRA C:2012 10.3 (Required)
Overflow

Overflow
Operation [conversion from int32 to int8] on scalar does not overflow in INT8 range
Conversion from int 32 to int 8
right: [-9 .. 9]
result: [-9 .. 9]
(result is truncated)

You can use the green overflow check as rationale to justify the rule violation.

Rules About Pointer Arithmetic

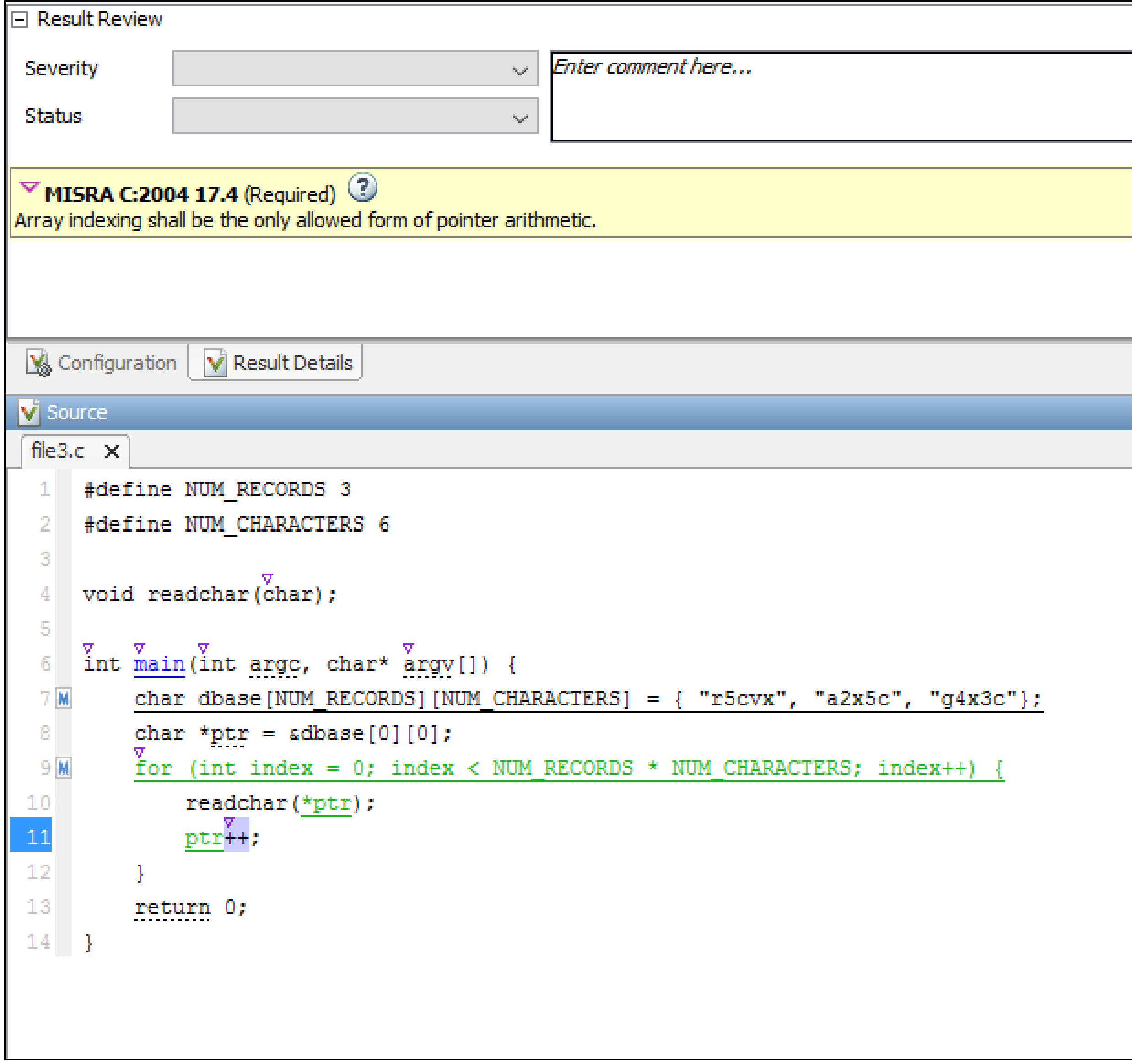
Pointer arithmetic on nonarray pointers are okay if the pointers stay within the allowed buffer.

In the following example, the operation `ptr++` violates MISRA C:2004 Rule 17.4. The rule states that array indexing shall be the only allowed form of pointer arithmetic. Here, a pointer that is not an array is incremented.

```
#define NUM_RECORDS 3
#define NUM_CHARACTERS 6

void readchar(char);

int main(int argc, char* argv[]) {
    char dbase[NUM_RECORDS][NUM_CHARACTERS] = { "r5cvx", "a2x5c", "g4x3c"};
    char *ptr = &dbase[0][0];
    for (int index = 0; index < NUM_RECORDS * NUM_CHARACTERS; index++) {
        readchar(*ptr);
        ptr++;
    }
    return 0;
}
```

Consider the rationale behind this rule. After an increment, a pointer can go outside the bounds of an allowed buffer (such as an array) or even point to an arbitrary location. Pointer arithmetic is fine as long as the pointer points within an allowed buffer. You can justify this particular rule violation by using the results of a Code Prover `Illegally dereferenced pointer` check.

Run Code Prover on this code. On the **Source** pane, click the `++` in `ptr++`. You see the expected violation of MISRA C:2004 Rule 17.4.



Click the * on the operation `readchar(*ptr)`. You see a green **Illegally dereferenced pointer** check. The green check indicates that the pointer points within allowed bounds when dereferenced.

Illegally dereferenced pointer
Pointer is within its bounds
Dereference of local pointer 'ptr' (pointer to int 8, size: 8 bits):
    Pointer is not null.
    Points to 1 bytes at offset [0 .. 17] in buffer of 18 bytes, so is within bounds (if memory is allocated).
    Pointer may point to variable or field of variable:
        'dbase', local to function 'main'.

You can use the green check to justify the rule violation.

See Also

Related Examples

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications”

Polyspace Results in Lines Containing Macros

Macros in C/C++ can improve readability and maintainability of code. A macro is a named fragment of code defined with the `#define` directive, for instance:

```
#define MAXSIZE 64
```

The macro name acts as a shorthand for the fragment of code. During preprocessing, each instance of a macro is replaced with its definition. For instance, in the above example, each time you use `MAXSIZE`, it is replaced with 64 during preprocessing.

Polyspace provides several conveniences for reviewing results in lines containing macros.

Macros in Source Lines Can Be Expanded in Place

If a source code line contains a macro, the **Source** pane displays the line with an icon on the left. You can click the icon to expand the macro, that is, see the macro definition, and click again to collapse the macro. See also:

- Bug Finder:
  - “Source Code in Polyspace Desktop User Interface”
  - “Source Code in Polyspace Access Web Interface”
- Code Prover:
  - “Source Code in Polyspace Desktop User Interface” on page 21-15
  - “Source Code in Polyspace Access Web Interface” on page 26-19

  If a macro expansion contains multiple Code Prover run-time checks, the line with the macro collapsed has the same color as the worst run-time check. See also “Code Prover Result and Source Code Colors” on page 32-2.

Results in Function-Like Macros Shown Only Once

A function-like macro is a macro that takes parameters, for instance:

```
#define max(x,y) x>y?x:y
```

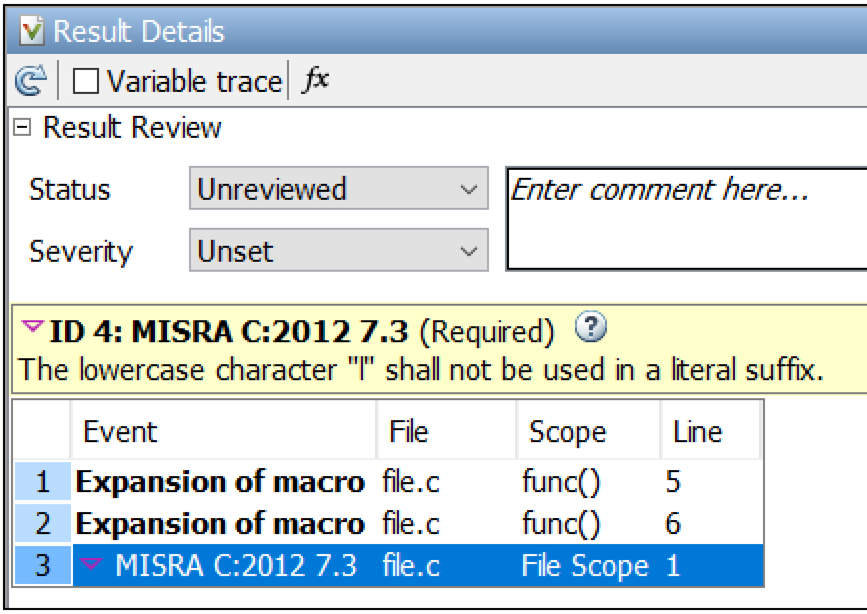
If a function-like macro causes a defect or coding standard violation, the result is displayed on the root cause of the issue: the macro parameter or the macro definition.

For instance:

- In this example, the definition of macro `LEFTOVER()` contains a lowercase `l` and violates `MISRA C:2012 Rule 7.3`. This result is shown on the macro definition.

  ```
  #define LEFTOVER(size) 10000ul - size   /* Noncompliant */
  #define REMAINDER(size) 10000UL - size  /* Compliant */

  void func(int arrSize, int arrCopySize) {
      int n = LEFTOVER(arrSize);
      int nCopy = LEFTOVER(arrCopySize);
      int m = REMAINDER(arrSize);
  }
  ```

The event list below the result message shows the instances where the macro is used. You can click on an **Expansion of macro** event to navigate to the macro usage in the source code.



- In this example, the definition of macro `COPY_ELEMENT()` results in an ambiguous evaluation order and violates `MISRA C:2012 Rule 13.2` only when the parameter `i++` is passed to it. This result is shown on the macro expansion, specifically on the parameter in the expansion.

```
int a[10], b[10];
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])

void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);         /* Compliant */
    COPY_ELEMENT (i++);       /* Noncompliant  */
}
```

This way of showing results in function-like macros enables you to easily fix them:

- For issues caused by the macro definition, you can implement the fix once. Tools that report on the macro expansion can show multiple violations for one root cause.

  In the preceding example, you can change the lowercase `l` in `LEFTOVER()` to fix the issue. The `REMAINDER()` macro shows this fix.
- For issues caused by the macro parameters, you can also implement the fix once.

  In the preceding example, you can compute `i++` in a separate step, and then pass `i` to the `COPY_ELEMENT()` macro to fix the issue.

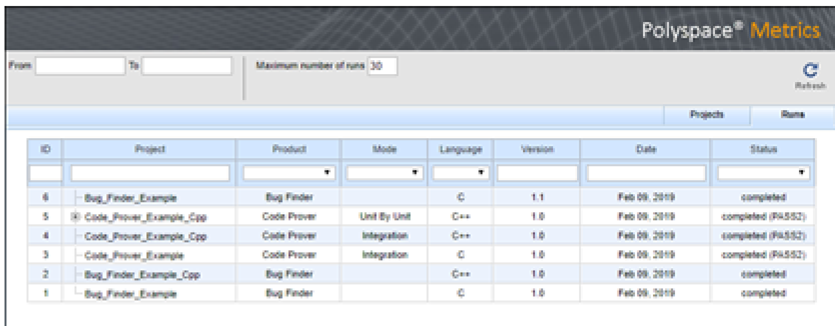
Migrate Results from Polyspace Metrics to Polyspace Access

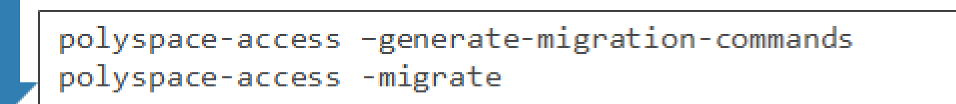
If you use Polyspace Metrics to store results and monitor the quality of your source code, you can transfer those results to Polyspace Access.

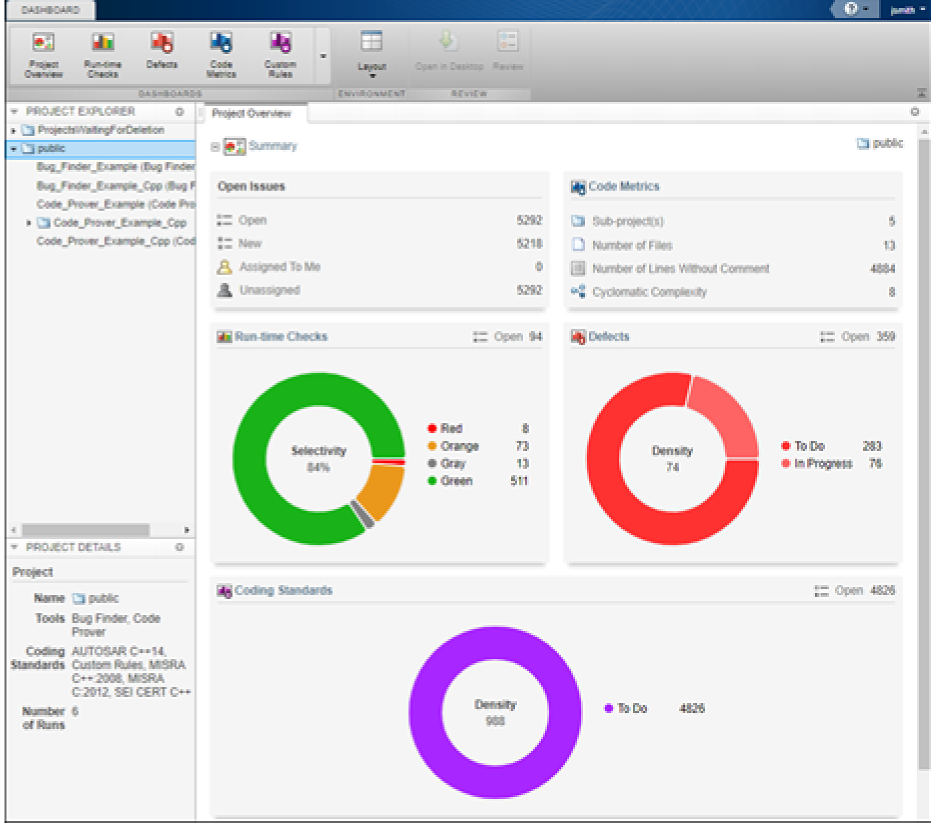
The Polyspace Access **DASHBOARD** perspective offers a web interface with navigation between projects and categories of results. From the **Project Overview** dashboard, view aggregated statistics for all your projects or drill down to view results details by category or file. For each family of findings, open an additional dashboard to see details. After you narrow down the set of findings that you want to address, open them in the **REVIEW** perspective to start reviewing individual results.

Note The **REVIEW** perspective is only available for analysis results generated with a Polyspace product version R2019a or later. To review R2018b or earlier results that you migrated to Polyspace Access, see “Open or Export Results from Polyspace Access” on page 29-2.

You can also review results from Polyspace Access by opening them in the Polyspace desktop interface. You do not need to download a local copy of Polyspace Access results to view those results in the desktop interface. The edits that you make to the results are saved directly in Polyspace Access and enable you to perform collaborative reviews.







Requirements for Migration

The transfer of results from the Metrics repository to the Polyspace Access database requires the `polyspace-access` binary. This binary is available under the *`polyspaceroot`*`/polyspace/bin` folder with a Polyspace installation. *`polyspaceroot`* is the Polyspace product installation folder, for instance `C:\Program Files\Polyspace Server\2019a`.

For syntax and examples, see `polyspace-access`.

Migration of Results

To migrate results from Polyspace Metrics to Polyspace Access, follow these steps. You must be logged in to your Metrics server to complete this operation.

1. Identify the Metrics results repository location. The Polyspace Metrics results are stored in the `results-repository` folder at that location.

   To view the path to this location, from the desktop interface, go to **Tools** > **Metrics Server Settings**. Or, at the command line, run the command `psqueue-check-config`.

   By default, results are stored under `C:\Users\`*`username`*`\AppData\Roaming\Polyspace_RLDatas\results-repository` on Windows and `/home/`*`username`*`/.polyspace/results-repository` on Linux. *`username`* is your computer login user name.
2. Generate migration scripts.

   Once you identify the folder of the repository from which you want to transfer results, define a migration strategy. You can choose to transfer all your projects or you can narrow down the scope of the transfer to a specific set of projects.

   Specify a set of projects with the options listed in this table.

Option	Description
`-max-project-runs` *`int`*	Number of most recent analysis runs you want to migrate for each project. For instance, to migrate only the last two analysis runs of a project, specify 2.
`-project-date-after` *`YYYY[-MM[-DD]]`*	Only migrate results that were uploaded to Polyspace Metrics on or after the specified date.
`-product` *`productName`*	Product used to analyze and produce project findings, specified as `bug-finder`, `code-prover`, or `polyspace-ada`.
`-analysis-mode` *`mode`*	Analysis mode used to generate project findings, specified as `integration` or `unit-by-unit`.

   For example, to transfer only Polyspace Bug Finder analysis results that you uploaded to Polyspace Metrics on or after June 2017, use this command:

```
polyspace-access -generate-migration-commands ^
C:\Users\username\AppData\Roaming\Polyspace_RLDatas\results-repository ^
-output-folder-path C:\Polyspace_Workspace\Migrate^
-project-date-after 2017-06^
-product bug-finder
```

   The command outputs a migration script file for each project stored in `C:\Users\`*`username`*`\AppData\Roaming\Polyspace_RLDatas\results-repository` that matches the specified product and date. The migration scripts are stored under `C:\Polyspace_Workspace\Migrate`.

   Before you continue, you can optionally open the migration scripts in a text editor and modify the `-project` or `-parent-project` parameters. The parameters correspond to the name of the project and the folder under which it is stored in Polyspace Access, respectively.

3 Migrate the projects.

After you generate the migration scripts, to transfer all the selected projects use those scripts with this migration command :

```
polyspace-access -host hostName -port port  ^
-migrate -option-file-path ^
C:\Polyspace_Workspace\Migrate
```

The command looks for migration scripts under `C:\Polyspace_Workspace\Migrate` and uploads the results to the Polyspace Access instance that you specify with *`hostName`*. Enter your Polyspace Access user name and password at the prompt.

`hostName` and *`port`* correspond to the host name and port number you specify in the URL of the Polyspace Access interface, for example `https://hostName:port/metrics/index.html`. If you are unsure about which host name and port number to use, contact your Polyspace Access administrator. Depending on your configuration, you might also need to specify the `-protocol` option in the migration command.

During the execution of a migration script, the command generates a temporary `STARTED` file. After each successful execution of a migration script, the command deletes the `STARTED` file and generates a corresponding `DONE` file in the same folder as the script. For example, the command generates `foo.started` during the execution of `foo.cmd`, and then `foo.done` once `foo.cmd` is done. Do not delete these `DONE` files until you have completed your migration from Metrics to Access.

Depending on the amount of data that you are transferring and on your network configuration, the migration might take a long time. You can interrupt the transfer, and then continue from where you left off at a later time. To stop the transfer, press **CTRL+C**. To restart the transfer:

a. Go to the folder where you store the migration scripts and open the `STARTED` file in a text editor. The file might be in a subfolder of the migration scripts folder.
b. Follow the instructions in the file, then reuse the same migration command that you used when you started the migration. The command skips projects that uploaded successfully.

If a project migration fails, go to the migration script folder. See the `FAILED` file for more information.

Differences in SQO Between Polyspace Metrics and Polyspace Access

After you migrate your projects from Polyspace Metrics to Polyspace Access, you might notice differences when you examine your code quality against "Evaluate Polyspace Code Prover Results Against Software Quality Objectives" on page 31-2.

The difference is due to the way Polyspace Metrics and Polyspace Access calculate the thresholds for the quality objectives. Polyspace Metrics looks at individual files to determine whether your code achieves a given SQO threshold. For instance, if file `foo.c` does not achieve threshold `SQO2`, then the whole project does not achieve that threshold.

Polyspace Access looks at the whole project to determine whether your source code meets a given SQO threshold. Even if file `foo.c` does not achieve the threshold, the whole project can still meet the specified quality objective threshold.

See Also

More About

- “Register Polyspace Desktop User Interface”
- “Upload Results to Polyspace Access” on page 2-28

Understanding Code Prover Results

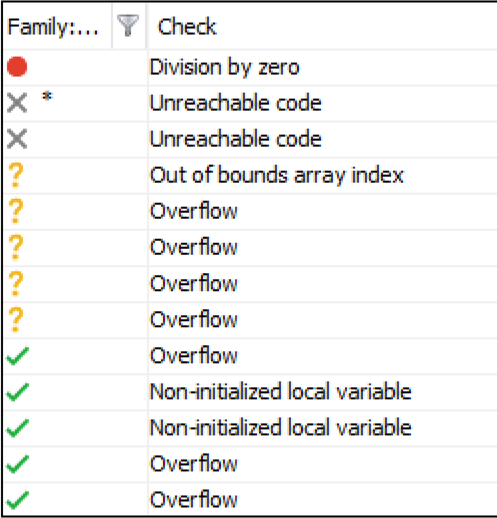
Code Prover Checks and Source Code Tooltips

Code Prover Result and Source Code Colors

This topic explains the various colors used in displaying the results of a Polyspace Code Prover analysis.

Result Colors

Polyspace displays the different verification results with specific icons and colors on the **Results List** and **Result Details** pane.



Run-Time Checks

Polyspace Code Prover checks each operation in your code for particular run-time errors. The software assigns a color to the operation based on whether it proved the existence or absence of a run-time error on all or some execution paths.

Check Color	Purpose	Example	Icon
Red	Highlights operations that are proven to cause a particular error on all execution paths*. Polyspace Code Prover verification determines errors with reference to the language standard. Though some of the errors can be acceptable for a particular compilation environment, they violate the language standard. To allow some of the environment-dependent behavior, use appropriate analysis options. For more information, see “Verification Assumptions” and “Check Behavior”.	Red **Overflow** on: `z = x+y;` The operation + overflows for every value of `x` and `y` that the verification considers at that point.	●

Check Color	Purpose	Example	Icon

Gray	Highlights unreachable code.	Gray **Unreachable code** check:
		
		`if(x>0)`
		`{}`
		`else`
		`{}`
		
		The `else` branch is unreachable for all values of `x` that the verification considers at that point.	

Orange	Highlights operations that can cause an error on certain execution paths. For more information, see “Orange Checks in Polyspace Code Prover” on page 33-2.	Orange **Overflow** on: `z = x+y;` The analysis could not prove whether the operation `+` overflows. The most common reason is that the operation overflows only for some values of `x` and `y` that the verification considers at that point. You can use the tooltips on the variables `x` and `y` in the operation to see the range of values that the verification considers.	
Green	Highlights operations that are proven to not cause a particular error on all execution paths*.	Green **Overflow** on: `z = x+y;` The operation `+` does not overflow for all values of `x` and `y` that the verification considers at that point.	

* For most checks, the software terminates an execution path following the first run-time error on the path. Therefore, if it proves a definite error (red) or absence of error (green) on an operation, the proof is valid only for the execution paths that have not yet been terminated at that point in the code. See “Code Prover Analysis Following Red and Orange Checks” on page 32-10.

Other Results

Besides checks for run-time errors, Polyspace Code Prover also displays other results about your code.

Result	Purpose	Icon
Coding rule violations	Indicates violation of predefined or custom coding rules.	for predefined rules and for custom rules.

Result	Purpose	Icon
Code metrics	Indicates code complexity metrics.	for metrics that do not exceed a limit you specified and ! for metrics that exceed a limit.
Global variables	Indicates global variable declaration.	? for shared potentially unprotected variables and X for non-shared unused variables

Source Code Colors

Polyspace uses the following color scheme for displaying code on the **Source** pane.

- *Lines with checks*:

  For every check on the **Results List** pane, Polyspace assigns the check color to the corresponding section of code.

  - For lines containing macros, if the macro is collapsed, then Polyspace colors the entire line with the color of the most severe check on the line. The severity decreases in this order: red, gray, orange, green.

    This unreachable `for` loop contains a macro `MAX_SIZE`. The entire line is colored gray.

    ```
    for (i = 0; i < MAX_SIZE; i++) {
    ```

    If there is no check in a line containing a macro, Polyspace underlines the line in black when the macro is collapsed.
  - For all other lines, Polyspace colors only the keyword or identifier associated with the check.

    This assignment has three checks: `i` and `used_global` are initialized but the array `tab` can be accessed outside its bounds. The `[` operator is colored orange to indicate the issue.

    ```
    tab[i] = used_global;
    ```

- *Lines with coding rule violations*:

  For every coding rule violation on the **Results List** pane, Polyspace assigns to the corresponding keyword or identifier:

  - A ▽ (inverted triangle) symbol if the coding rule is a predefined rule. The predefined rules available are MISRA C, MISRA AC AGC, MISRA C++, or JSF C++.

    This `if` statement and `||` operation violates MISRA rules.

    ```
    if (x < 0 || x > 20) return -1;
    ```

  - A ▼ symbol if the coding rule is a custom rule.

    This function name violates a custom naming convention.

    ```
    int polynomia(int input)
    ```

- *Lines with tooltips*:

  If a tooltip is available for a keyword or identifier on the **Source** pane, Polyspace:

  - Uses solid underlining for the keyword or identifier if it is associated with a check.

    This line has both checks and tooltips on `input`, `%` and `used_global`.

    ```
    result = input % used_global;
    ```

  - Uses dashed underlining for the keyword or identifier if it is not associated with a check.

    This line has tooltips on `for` and `<`, but no checks on them.

    ```
    for (i = 0; i < 10; i++)
    ```

  - Uses dashed red underlining on function calls or loop commands to indicate that the function body or loop body contains a potential run-time error. The tooltip shows the line in the function or loop body that causes the error.

    This call to `function_with_red` leads to a run-time error.

    ```
    i = function_with_red(0);
    ```

- *Function definitions*:

  When a function is defined, Polyspace colors the function name in blue.

  ```
  void task1(void) {
  ```

- *Lines deactivated due to conditional compilation*:

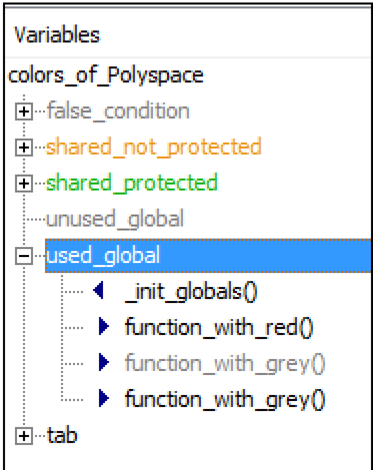
  Polyspace assigns a light gray color to code that is preprocessed out due to conditional compilation. Such code occurs, for instance, in `#ifdef` statements where the macro for a branch is not defined. This code does not affect the verification (or actual runtime behavior).

  ```
  #ifdef ACTIVE
          /* this code is not processed! */
          tab[0] = used_global;
  ```

Global Variable Colors

The **Variable Access** pane shows the global variables in your code along with the read and write operations on the variables.

For instance, `used_global` is a global variable that is accessed four times: once during initialization, once in the function `function_with_red`, and twice in the function `function_with_grey`.



The color scheme is as follows:

- *Variable colors*:
  - Orange: Shared, unprotected global variable (only applicable to multitasking code)
  - Green: Shared, protected global variable (only applicable to multitasking code)
  - Black: Unshared, used global variable
  - Gray: Unshared, unused global variable

  See “Global Variables”.
- *Operation colors*: If an operation occurs in unreachable code, it is grey, otherwise black.

  In the preceding example, one operation in the function `function_with_grey` is unreachable but the other is reachable.

For more information, see “Variable Access in Polyspace Desktop User Interface” on page 21-27.

See Also

Related Examples

- “Orange Checks in Polyspace Code Prover” on page 33-2
- “Code Prover Analysis Following Red and Orange Checks” on page 32-10

Reviewing Code Prover Run-Time Checks

Polyspace Code Prover checks each operation in your C/C++ code for certain run-time errors and displays the result as a red, green or orange check. For more information, see “Code Prover Result and Source Code Colors” on page 32-2.

You must review a red or orange check and determine whether to fix your code. The tables below list the checks that Polyspace Code Prover performs and how you can review them.

Data Flow Checks

The following table outlines how to review the results of some of the data flow checks in Code Prover.

Check	How to Review	Details
`Function not called`	Investigate why a function does not appear in the call graph starting from the `main` or another entry point function.	“Review and Fix Function Not Called Checks” on page 22-11
`Function not reachable`	Identify the call sites of a function and investigate why they occur in unreachable code.	“Review and Fix Function Not Reachable Checks” on page 22-13
`Non-initialized local variable`	Locate prior variable initializations if any and see if your program can bypass them.	“Review and Fix Non-initialized Local Variable Checks” on page 22-35
`Non-initialized pointer`	Locate prior pointer initializations if any and see if your program can bypass them.	“Review and Fix Non-initialized Pointer Checks” on page 22-38
`Non-initialized variable`	Locate prior initializations of the global variable if any and see if your program can bypass them.	“Review and Fix Non-initialized Variable Checks” on page 22-40
`Return value not initialized`	Identify paths through your function body that do not end in a `return` statement.	“Review and Fix Return Value Not Initialized Checks” on page 22-63
`Unreachable code`	Investigate why a conditional statement in your code is redundant, for instance, always true or always false.	“Review and Fix Unreachable Code Checks” on page 22-68

Numerical Checks

The following table outlines how to review the results of some of the numerical checks in Code Prover.

Check	How to Review	Details
`Division by zero`	Review prior operations in your code that lead to zero value of a denominator.	“Review and Fix Division by Zero Checks” on page 22-7

Check	How to Review	Details
`Invalid shift operations`	Review prior operations in your code that lead to a shift amount outside bounds or a negative value being left-shifted.	"Review and Fix Invalid Shift Operations Checks" on page 22-26
`Overflow`	Review prior operations in your code that lead to an operation overflowing.	"Review and Fix Overflow Checks" on page 22-57

Static Memory Checks

The following table outlines how to review the results of some of the static memory checks in Code Prover.

Check	How to Review	Details
`Absolute address usage`	Review uses of absolute address in your code and make sure that the addresses are valid.	"Review and Fix Absolute Address Usage Checks" on page 22-2
`Illegally dereferenced pointer`	Review prior operations in your code that lead to a pointer pointing outside its allocated memory buffer.	"Review and Fix Illegally Dereferenced Pointer Checks" on page 22-17
`Out of bounds array index`	Review prior operations in your code that lead to an array index being greater than or equal to array size.	"Review and Fix Out of Bounds Array Index Checks" on page 22-53

Control Flow Checks

The following table outlines how to review the results of some of the control flow checks in Code Prover.

Check	How to Review	Details
`Non-terminating call`	Review operations in the function body and find which run-time error occurs because of issues specific to the current function call.	"Review and Fix Non-Terminating Call Checks" on page 22-42
`Non-terminating loop`	Review operations in the loop and determine why the loop does not terminate or why a definite run-time error occurs in one of the loop runs.	"Review and Fix Non-Terminating Loop Checks" on page 22-46

C++ Checks

The following table outlines how to review the results of some of the C++-specific checks in Code Prover.

Check	How to Review	Details
`Invalid C++ specific operations`	Determine root cause of nonpositive array size or incorrect usage of the `typeid` or the `dynamic_cast` operator.	"Review and Fix Invalid C++ Specific Operations Checks" on page 22-24
`Function not returning value`	Identify paths through your function body that do not end in a `return` statement.	"Review and Fix Function Not Returning Value Checks" on page 22-15
`Incorrect object oriented programming`	Investigate why a certain `virtual` member call or `this` pointer usage represents an incorrect pattern of object oriented programming.	"Review and Fix Incorrect Object Oriented Programming Checks" on page 22-22
`Null this-pointer calling method`	Investigate why the pointer to the current object can be `NULL`-valued.	"Review and Fix Null This-pointer Calling Method Checks" on page 22-51
`Uncaught exception`	Investigate how an exception can escape uncaught from the function where it is thrown.	"Review and Fix Uncaught Exception Checks" on page 22-66

Other Checks

The following table outlines how to review the results of some of the uncategorised checks in Code Prover.

Check	How to Review	Details
`Correctness condition`	Find the root cause of a function pointer misuse, incorrect array conversion or variable values outside specified constraints.	"Review and Fix Correctness Condition Checks" on page 22-3
`Invalid use of standard library routine`	Investigate why the arguments in the current call to the standard library routine are invalid.	"Review and Fix Invalid Use of Standard Library Routine Checks" on page 22-30
`User assertion`	Investigate why the condition in an `assert` statement fails.	"Review and Fix User Assertion Checks" on page 22-73

See Also

Related Examples

- "Code Prover Result and Source Code Colors" on page 32-2
- "Run-Time Checks"

Code Prover Analysis Following Red and Orange Checks

Polyspace Code Prover checks for run-time errors in C/C++ code. The analysis considers that all execution paths that contain a run-time error terminate at the location of the error. For a given execution path, Polyspace highlights the first occurrence of a run-time error as a red or orange check and excludes that path from consideration. Therefore:

- Following a red check, Polyspace does not analyze the remaining code in the same scope as the check.
- Following an orange check, Polyspace analyzes the remaining code. But it considers only a reduced subset of execution paths that did not contain the run-time error. Therefore, if a green check occurs on an operation *after an orange check*, it means that the operation does not cause a run-time error only for this reduced set of execution paths.

The path containing a run-time error is terminated for the following reasons:

- The state of the program is unknown following the error. For instance, following an Illegally dereferenced pointer error on an operation `x=*ptr`, the value of `x` is unknown.
- You can review an error as early in your code as possible, because the first error on an execution path is shown in the verification results.
- You do not have to review and then fix or justify the same result more than once. For instance, consider these statements, where the vertical ellipsis represents code in which the variable `i` is not modified.

  ```
  x = arr[i];
  .
  .
  y = arr[i];
  ```

  If an orange Out of bounds array index check appears on `x=arr[i]`, it means that `i` can be outside the array bounds. You do not want to review another orange check on `y=arr[i]` arising from the same cause.

Exceptions to the path-termination behavior can occur when you use specific options. For instance:

- For an overflow, if you specify `warn-with-wrap-around` or `allow` for `Overflow mode for signed integer (-signed-integer-overflows)` or `Overflow mode for unsigned integer (-unsigned-integer-overflows)`, Polyspace wraps the result of an overflow and does not terminate the execution paths.
- For a subnormal float result, if you specify `warn-all` for `Subnormal detection mode (-check-subnormal)`, Polyspace does not terminate the execution paths with subnormal results.

The path-termination behavior leads to results of checks being dependent on previous checks in the same scope. The following examples show how the path termination can result in checks that can be misleading when viewed out of context. Understand the examples below thoroughly to practice reviewing checks in context of the remaining code.

Code Following Red Check

The following example shows what happens after a red check:

```
void red(void)
{
```

```
int x;
x = 1 / x ;
x = x + 1;
}
```

When Polyspace verification reaches the division by `x`, `x` has not yet been initialized. Therefore, the software generates a red `Non-initialized local variable` check for `x`.

Execution paths beyond division by `x` are stopped. No checks are generated for the statement `x = x + 1;`.

Green Check Following Orange Check

The following example shows how a green check can result from a previous orange check. An orange check terminates execution paths that contain an error. A green check on an operation after an orange check means that the operation does not cause a run-time error only for the remaining execution paths.

```
extern int Read_An_Input(void);
void propagate(void)
{
    int x;
    int y[100];
    x = Read_An_Input();
    y[x] = 0;
    y[x] = 0;
}
```

In this function:

- `x` is assigned the return value of `Read_An_Input`. After this assignment, the software estimates the range of `x` as `[-2^31, 2^31-1]`.
- The first `y[x]=0;` shows an `Out of bounds array index` error because `x` can have negative values.
- After the first `y[x]=0;`, from the size of `y`, the software estimates `x` to be in the range `[0,99]`.
- The second `y[x]=0;` shows a green check because `x` lies in the range `[0,99]`.

Gray Check Following Orange Check

The following example shows how a gray check can result from a previous orange check.

Consider the following example:

```
extern int read_an_input(void);

void main(void)
{
    int x;
    int y[100];
    x = read_an_input();
    y[x] = 0;
    y[x-1] = (1 / x) + x ;
    if (x == 0)
        y[x] = 1;
}
```

From the gray check, you can trace backwards as follows:

- The line `y[x]=1;` is unreachable.
- Therefore, the test to assess whether `x = 0` is always false.
- The return value of `read_an_input()` is never equal to 0.

However, `read_an_input` can return any value in the full integer range, so this is not the correct explanation.

Instead, consider the execution path leading to the gray code:

- The orange **Out of bounds array index** check on `y[x]=0;` means that subsequent lines deal with `x` in `[0,99]`.
- The orange **Division by Zero** check on the division by `x` means that `x` cannot be equal to 0 on the subsequent lines. Therefore, following that line, `x` is in `[1,99]`.
- Therefore, `x` is never equal to 0 in the `if` condition. Also, the array access through `y[x-1]` shows a green check.

Red Check Following Orange Check

The following example shows how a red error can reveal a bug which occurred on previous lines.

```
%% file1.c %%

void f(int);
int read_an_input(void);

int main()  {
    int x,old_x;
    x = read_an_input();
    old_x = x;
    if (x<0 || x>10)
      return 1;
    f(x);
    x = 1 / old_x;
    // Red Division by Zero
    return 0;
}
```

```
%% file2.c %%

#include <math.h>

void f(int a) {
    int tmp;
    tmp = sqrt(0-a);
}
```

A red check occurs on `x=1/old_x;` in `file1.c` because of the following sequence of steps during verification:

1. When `x` is assigned to `old_x` in `file1.c`, the verification assumes that `x` and `old_x` have the full range of an integer, that is `[-2^31 , 2^31-1]`.
2. Following the `if` clause in `file1.c`, `x` is in `[0,10]`. Because `x` and `old_x` are equal, Polyspace considers that `old_x` is in `[0,10]` as well.
3. When `x` is passed to `f` in `file1.c`, the only possible value that `x` can have is 0. All other values lead to a run-time exception in `file2.c`, that is `tmp = sqrt(0–a);`.
4. A red error occurs on `x=1/old_x;` in `file1.c` because the software assumes `old_x` to be 0 as well.

Red and Green Checks in Unreachable Code

Code Prover can sometimes show red and green checks in code that is supposed to be unreachable and gray. When propagating variable ranges, Code Prover sometimes makes approximations. In

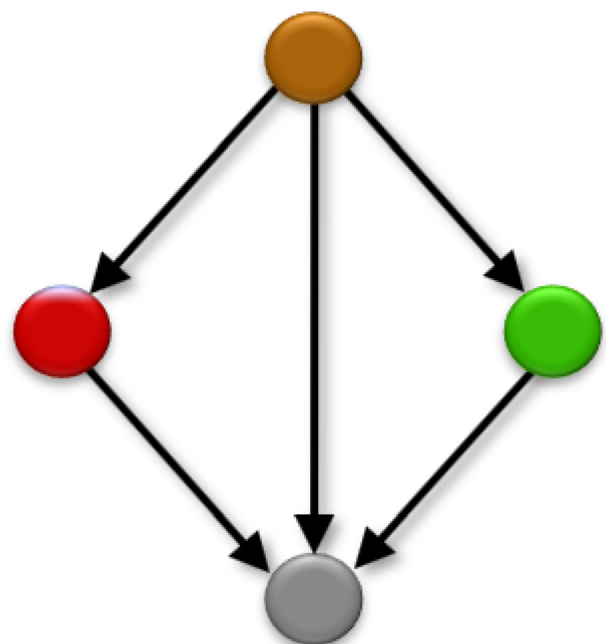
making these approximations, Code Prover might consider an otherwise unreachable branch as reachable. If a check fails in that unreachable branch, it is colored red and if it passes, it is colored green.

Consider the statement:

```
if (test_var == 5) {
  // Code Section
}
```

If `test_var` does not have the value 5, the `if` branch is unreachable. If Code Prover makes an approximation because of which `test_var` acquires the value 5, the branch is now reachable and can show checks of other colors.

Use this figure to understand the effect of approximations. Because of approximations, a check color that is higher up can supersede the colors below. A check that should be red or green (indicating a definite error or definite absence of error) can become orange because a variable acquires extra values that cannot occur at run time. A check that should be gray can show red, green and orange checks because Code Prover considers an unreachable branch as reachable. Because of the approximations, Code Prover might not show all unreachable sections of code.



See Also

Related Examples

- “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2
- “Order of Code Prover Run-Time Checks” on page 32-15

- "Code Prover Result and Source Code Colors" on page 32-2

Order of Code Prover Run-Time Checks

If multiple checks are performed on the same operation, Code Prover performs them in a specific order. The order of checks is important only if one of the checks is not green. If a check is red, the subsequent checks are not performed. If a check is orange, the subsequent checks are performed for a reduced set of values. For details, see "Code Prover Analysis Following Red and Orange Checks" on page 32-10.

A simple example is the order of checks on a pointer dereference. Code Prover first checks if the pointer is initialized and then checks if the pointer points to a valid location. The check `Illegally dereferenced pointer` follows the check `Non-initialized local variable`.

If one of the operands in a binary operation is a floating-point variable, Code Prover performs these checks on the operation in this order:

1. `Invalid operation on floats`: If you enable the option to consider non-finite floats, this check determines if the operation can result in NaN.
2. `Overflow`: This check determines if the result overflows.

   If you enable the option to consider non-finite floats, this check determines if the operation can result in infinities.
3. `Subnormal float`: If you enable the subnormal detection mode, this check determines if the operation can result in subnormal values.

For instance, suppose you enable options to forbid infinities, NaNs and subnormal results. In this example, the operation `y = x + 0;` is checked for all three issues. The operation appears red but consists of three checks: an orange **Invalid operation on floats**, an orange **Overflow**, and a red **Subnormal float** check.

```
#include <float.h>
#include <assert.h>

double input();

int main() {
  double x = input();
  double y;
  assert (x != x || x > DBL_MAX  || (x > 0. && x < DBL_MIN));
  y = x + 0.;
  return 0;
}
```

At the + operation, `x` can have three groups of values: `x` is NaN, `x > DBL_MAX`, and `x > 0. && x < DBL_MIN`.

The checks are performed in this order:

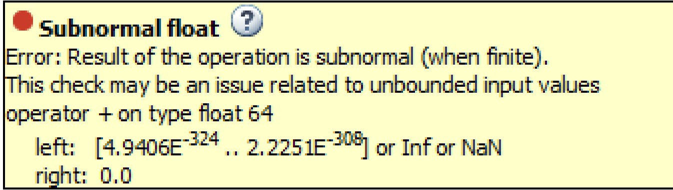
1. **Invalid operation on floats**: The check is orange because one execution path considers that `x` can be `NaN`.

   For the next checks, this path is not considered.
2. **Overflow**: The check is orange because one group of execution paths considers that `x > DBL_MAX`. For this group of paths, the + operation can result in infinity.

   For the next check, this group of paths is not considered.

3 **Subnormal float**: On the remaining group of execution paths, `x > 0. && x < DBL_MIN`. All values of `x` cause subnormal results. Therefore, this check is red.

The message on the **Result Details** pane reflects this reduction in paths. The message for the **Subnormal float** check states `(when finite)` to show that infinite values were removed from consideration.



The values for the left and right operands reflect all values before the operation, and not the reduced set of values. Therefore, the left operand still shows `Inf` and `NaN` even though these values were not considered for the check.

See Also

`Consider non finite floats (-allow-non-finite-floats)` | `Infinities (-check-infinite)` | `NaNs (-check-nan)` | `Overflow` | `Invalid operation on floats` | `Subnormal float`

More About

- “Code Prover Analysis Following Red and Orange Checks” on page 32-10

Variable Ranges in Source Code Tooltips After Code Prover Analysis

Polyspace Code Prover analyzes C/C++ code for run-time errors and reports the analysis results as checks on operations. The checks are colored green, red, or orange based on whether they pass, fail or remain inconclusive. In addition to the checks, the analysis also reports ranges on variables in tooltips throughout the source code. These range tooltips can support investigation of the reported run-time checks and also facilitate a deeper understanding of the code when used with other navigation tools in the Polyspace user interface or Polyspace Access web interface.

For instance, this tooltip on the variable `beta` indicates that the variable is a local variable, has data type `float` (on a target where `float` has 32 bits) and has a value in the range [0.1 .. 0.5]. The range combines values of `beta` from all possible execution paths that pass through that code operation.

Why Code Prover Reports Ranges on Variables

Code Prover analyzes each operation in the source code for all possible execution paths through the operation (subject to verification assumptions). When the analysis reports a range on an instance of a variable or an operation, the range combines values from all execution paths that lead to the variable or operation.

Consider the following example. When Code Prover analyzes the function `func`, the analysis reports variable ranges on almost all variables in the function. The code comments on each line show the variable ranges reported in a Code Prover analysis. Note that the ranges shown are the ones that occur *after* the operations on the line.

```
int func(unsigned int count) { // count is in [0 .. UINT_MAX]
    int var;
    if(count <= 5)
        var = count; //var is in [0..5]
    else
        var = 100;
    return var; //var is 100 or in [0..5]
}
```

For instance, you can see variable ranges in tooltips:

- In the beginning of the function.

  Suppose that `func` is not called elsewhere in the code. Based on the data type `unsigned int`, Code Prover assumes that `count` can have values in the range [0 .. 2^{32} - 1] or [0 .. `UINT_MAX`].
- In each branch of a conditional statement.

  An execution path can enter the `if` branch of the `if-else` statement only if `count` is less than or equal to 5. Therefore, inside the `if` block, `count` can have values only in the range [0 .. 5]. The variable `var`, which gets its value from the constrained `count`, also has the same constraint.
- At the end, when the function returns.

  In the `return` statement, `var` can have either the value [0 .. 5] from the `if` branch or the value 100 from the `else` branch. The tooltip on `var` in the `return` statement merges these two ranges and shows the range, 100 or [0 .. 5].

Using the tooltips on variable ranges, you can track the data flow in your code and understand how a variable acquires a value that could lead to a run-time error.

Why Variable Ranges Can Sometimes Be Narrower Than Expected

Sometimes, the range reported on a variable can be narrower than what you expect. A narrower-than-expected range can mean that two seemingly unrelated variables might be related from a previous operation.

Consider this example. The code comments on each line show the variable ranges reported in a Code Prover analysis *at the end of the operations on the line*.

```
void func(int arg) {

    int first, second, diff;
    first = arg;
    assert( first >= 0 && first <= 256*400); // first is in [0 .. 102400]
    second = (first << 4);  // second is in [0 .. 1638400]
    diff = (first * 16) -  (second + 256);  // diff is only -256
}
```

At first glance, the tooltip on `diff` in the last line might be surprising. The variable `first` is in the range [0..102400] and the variable `second` is in the range [0 ..1638400], but the difference `diff` has only one value, `-256`.

The reason for the single value of `diff` is that the previous operation:

```
second = (first << 4);
```

relates `first` and `second` in such a way that `first * 16` is always equal to `second`, irrespective of the values of `first` and `second`. Therefore, they cancel each other on all execution paths, leading to a single value of `diff`.

If you see narrower-than-expected ranges in your code, look for previous operations that might relate two of the variables involved in the current operation. You can also enter the pragma `Inspection_Point` before an operation and then analyze the code to identify a relation between two of the variables in an operation. See “Find Relations Between Variables in Code” on page 22-77.

See Also

More About

- “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2
- “Interpret Code Prover Results in Polyspace Access Web Interface” on page 26-2
- “Find Relations Between Variables in Code” on page 22-77
- “Code Prover Assumptions About Variable Ranges From Data Types”
- “Why Polyspace Verification Uses Approximations”

Orange Checks in Polyspace Code Prover

Orange Checks in Polyspace Code Prover

In this section...

"When Orange Checks Occur" on page 33-2
"Why Review Orange Checks" on page 33-3
"How to Review Orange Checks" on page 33-3
"How to Reduce Orange Checks" on page 33-3

Polyspace Code Prover checks each operation in your C/C++ code for certain run-time errors and displays the result as a red, green or orange check. For more information, see "Code Prover Result and Source Code Colors" on page 32-2. This topic describes orange checks in the results of a Polyspace Code Prover analysis.

When Orange Checks Occur

An orange check indicates that Polyspace detects a possible run-time error but cannot prove it. A check on an operation appears orange if both conditions are true:

First condition	Second condition	Example
The operation occurs multiple times on an execution path or on multiple execution paths.	During static verification, the operation fails only a fraction of times or only on a fraction of paths.	The operation occurs in: • A loop with more than one iterations. • A function that is called more than once.
The operation involves a variable that can take multiple values.	During static verification, the operation fails only for a fraction of values.	The operation involves a `volatile` variable.

During static verification, Polyspace can consider more execution paths than the execution paths that occur during run time. If an operation fails on a subset of paths, Polyspace cannot determine if that subset actually occurs during run time. Therefore, instead of a red check, it produces an orange check on the operation.

Orange Checks from Multiple Paths

Consider this example:

```
void main() {
  func(1);
  func(0);
}

double func(int value) {
  return (1.0/value); //Orange check
}
```

`func` is called twice with two arguments. Only one of the calls results in a division by zero in the body of `func`. Code Prover shows this result as an orange **Division by zero** check.

Orange Checks from Multiple Values

Consider this example:

```
double func(int value) {
  int reducedValue = value%21 - 10; // Result in [-10,10]
  return 1.0/reducedValue; //Orange check
}
```

If the call context of `func` is unknown, Code Prover assumes that its argument `value` can take any `int` value. As a result, `reducedValue` can take any value in [-10,10]. One of these values is zero, which causes a division by zero in `func`. Code Prover shows this result as an orange **Division by zero** check.

Why Review Orange Checks

Considering a superset of actual execution paths is a sound approximation because Polyspace does not lose information. If an operation contains a run-time error, Polyspace does not produce a green check on the operation. If Polyspace cannot prove the run-time error because of approximations, it produces an orange check. Therefore, you must review orange checks.

Examples of Polyspace approximations include:

- Approximating the range of a variable at a certain point in the execution path. For instance, Polyspace can approximate the range `{-1} U [0,10]` of a `float` variable by `[-1,10]`.
- Approximating the interleaving of instructions in multitasking code. For instance, even if certain instructions in a pair of tasks cannot interrupt each other, Polyspace verification might not take that into account.

How to Review Orange Checks

To ensure that an operation does not fail during run time:

1. Determine if the execution paths on which the operation fails can actually occur.

   For more information, see “Interpret Code Prover Results in Polyspace Desktop User Interface” on page 21-2.
2. If any of the execution paths can occur, fix the cause of the failure.
3. If the execution paths cannot occur, enter a comment in your Polyspace result or source code, describing why they cannot occur. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2.

   In a later verification, you can import these comments into your results. Then, if the orange check persists in the later verification, you do not have to review it again.

How to Reduce Orange Checks

Polyspace performs approximations because of one of the following.

- Your code does not contain complete information about run-time execution. For example, your code is partially developed or contains variables whose values are known only at run time.

If you want fewer orange checks, provide the information that Polyspace requires. For more information, see “Provide Context for Verification” on page 33-17.

- Your code is very complex. For example, there can be multiple type conversions or multiple `goto` statements.

  If you want fewer orange checks, reduce the complexity of your code and follow recommended coding practices. For more information, see “Follow Coding Rules” on page 33-18.
- Polyspace must complete the verification in reasonable time and use reasonable computing resources.

  If you want fewer orange checks, improve the verification precision. But higher precision also increases verification time. For more information, see “Improve Verification Precision” on page 33-18.

See Also

More About

- “Managing Orange Checks in Polyspace Code Prover” on page 33-5
- “Critical Orange Checks in Polyspace Code Prover” on page 33-9
- “Reduce Orange Checks in Polyspace Code Prover” on page 33-17
- “Limit Display of Orange Checks in Polyspace Desktop User Interface” on page 33-14

Managing Orange Checks in Polyspace Code Prover

Polyspace checks every operation in your code for certain run-time errors. Therefore, you can have several orange checks in your verification results. To avoid spending unreasonable time on an orange check review, you must develop an efficient review process.

Depending on your stage of software development and quality goals, you can choose to:

- Review all red checks and critical orange checks. See “Critical Orange Checks in Polyspace Code Prover” on page 33-9.
- Review all red checks and all orange checks.

Software Development Stage

<table>
<tr><th>Development Stage</th><th>Situation</th><th>Review Process</th></tr>
<tr><td>Initial stage or unit development stage</td><td>In initial stages of development, you can have partially developed code or want to verify each source file independently. In that case, it is possible that:
You have not defined all your functions and class methods.
You do not have a <code>main</code> function
Because of insufficient information in the code, Polyspace makes assumptions that result in many orange checks. For instance, if you use the default configuration, Polyspace assumes full range for inputs of functions that are not called in the code.</td><td>In the initial stages of development, review all red checks. For orange checks, depending on your requirements, do one of the following:
You want your partially developed code to be free of errors independent of the remaining code. For instance, you want your functions to not cause run-time errors for any input.

If so, review orange checks at this stage.
You might want your partially developed code to be free of errors only in the context of the remaining code.

If so, do one of the following:
Ignore orange checks at this stage.
Provide the context and then review orange checks. For instance, you can provide stubs for undefined functions to emulate them more accurately.

For more information, see “Provide Context for Verification” on page 33-17.</td></tr>
<tr><td>Later stage or integration stage</td><td>In later stages of development, you have provided all your source files. However, it is possible that your code does not contain all information required for verification. For example, you have variables whose values are known only at run time.</td><td>Depending on the time you want to spend, do one of the following:
Review red checks only.
Review red and critical orange checks.</td></tr>
</table>

<table>
<tr><th>Development Stage</th><th>Situation</th><th>Review Process</th></tr>
<tr><td>Final stage</td><td>You have provided all your source files.
You have emulated run-time environment accurately through the verification options.</td><td>Depending on the time you want to spend, do one of the following:
Review red checks and critical orange checks.
Review red checks and all orange checks.
For each orange check:
If the check indicates a run-time error, fix the cause of the error.
If the check indicates a Polyspace approximation, enter a comment in your results or source code.
As part of your final release process, you can have one of these criteria:
All red and critical orange checks must be reviewed and justified.
All red and orange checks must be reviewed and justified.
To justify a check, assign the Status of <code>No action planned</code> or <code>Justified</code> to the check.</td></tr>
</table>

Quality Goals

For critical applications, you must review all red and orange checks.

- If an orange check indicates a run-time error, fix the cause of the error.
- If an orange check indicates a Polyspace approximation, enter a comment in your results or source code.

As part of your final release process, review and justify all red and orange checks. To justify a check, assign the **Status** of `No action planned` or `Justified` to the check.

For noncritical applications, you can choose whether or not to review the noncritical orange checks.

See Also

Related Examples

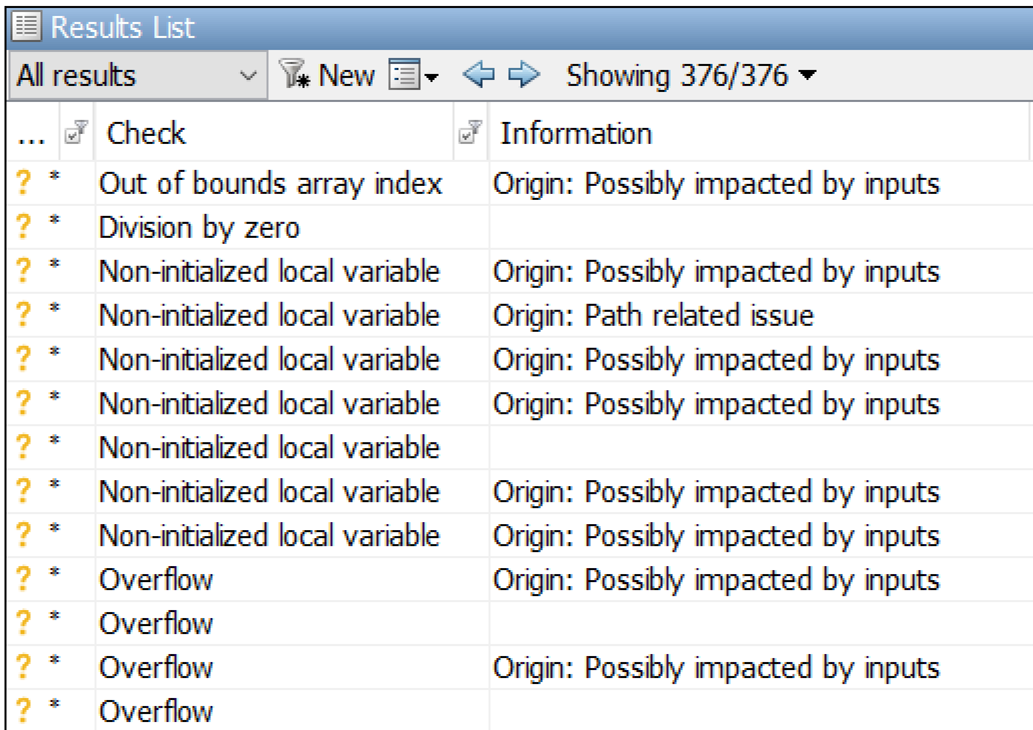
- "Limit Display of Orange Checks in Polyspace Desktop User Interface" on page 33-14

More About

- "Orange Checks in Polyspace Code Prover" on page 33-2

Critical Orange Checks in Polyspace Code Prover

Code Prover exhaustively checks your code for certain types of run-time errors. If a check is inconclusive, the result is displayed in orange (with a question mark). With each orange check, the analysis provides additional information that can help determine whether the check represents a possible run-time error or the result of an overapproximation or a broad assumption.



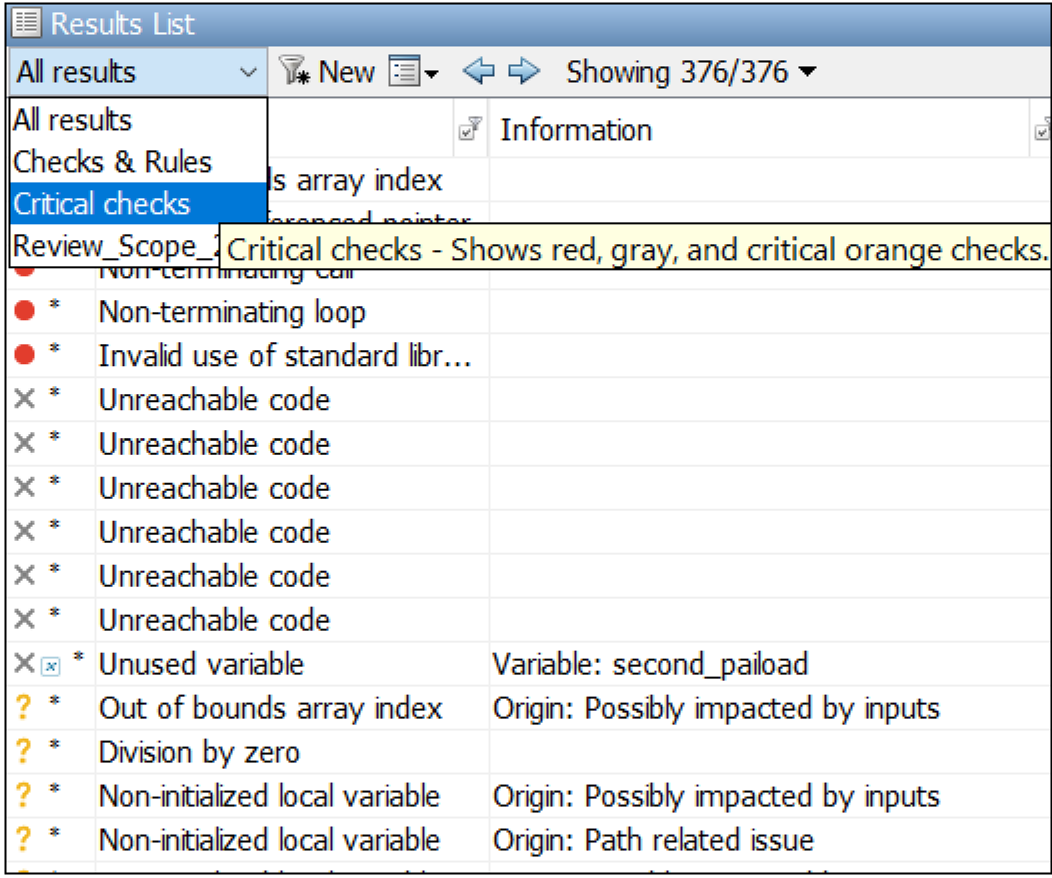
If a check is more likely to be a run-time error, it is considered as a critical orange check. Note that red and grey checks fall under the purview of critical checks by definition, since a red check indicates a definite run-time error and a grey check indicates code that is definitely unreachable (under the current verification assumptions). There is typically no question of determining likelihood when reviewing errors denoted with red and grey checks. With orange checks, critical checks help to focus review on results that are more likely to be run-time errors.

How to See Only Critical Checks

You can reduce the list of Code Prover results to only red, grey and critical orange checks.

Polyspace Desktop User Interface

In the user interface of the Polyspace desktop products, you can directly focus the list of results to critical checks only. On the **Results List** pane, from the **All results** dropdown, select **Critical checks**.

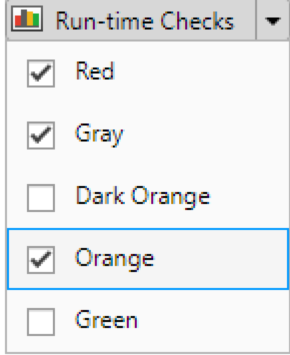


Selecting **Critical checks** retains only red, grey and critical orange checks on the **Results List** pane.

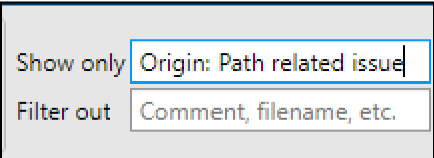
Polyspace Access Web Interface

In the Polyspace Access web interface, you can use separate filters for red, grey and critical orange checks:

1. Display only red, grey and orange checks using the color-based filters.



2. Display only critical orange checks using the text-based filters. In the **Show only** text filter, enter **Origin: Path related issue** to see path-related orange checks and **Origin: Bounded inputs** to see orange checks related to bounded inputs.



Which Orange Checks Are Considered Critical

The notion of critical checks allows you to prioritize your review of orange checks. You can choose to review the critical orange checks before other orange checks, or review only the critical orange checks. See also “Managing Orange Checks in Polyspace Code Prover” on page 33-5.

The critical orange checks are related to *path* and *bounded input values*. Here, input values refer to values that are external to the application. Examples include:

- Inputs to functions called by generated main. For more information on functions called by generated main, see `Functions to call (-main-generator-calls)`.
- Global and volatile variables.
- Data returned by a stubbed function. The data can be the value returned by the function or a function parameter modified through a pointer.

Note that there might be orange checks where the software cannot determine whether the check is path-related or input-related. Therefore, a safer approach might be to review critical checks before others but not confine the review to critical checks only.

Path

Checks that are path-related fall under the purview of critical orange checks. On the **Results List** pane, for these checks, the **Information** column contains the entry, **Origin: Path related issue**. These checks are critical because the software has identified distinct execution paths where a run-time error is likely to occur. The execution paths are distinct and separable from other paths that end up at the line with the check because they pass through different instructions in your code (another possibility would have been passing through the same instructions but for different inputs).

The following example shows a path-related orange check.

```
void path(int x) {
   int result;
   result = 1 / (x - 10);
   // Orange division by zero
 }

void main() {
    path(1);
    path(10);
 }
```

The software identifies the orange **Division by zero** check as a potential error. The **Result Details** pane indicates the potential error:

```
...
Warning: scalar division by zero may occur
...
```

This **Division by zero** check on `result=1/(x-10)` is orange because:

- `path(1)` does not cause a division by zero error.
- `path(10)` causes a division by zero error.

Note that the orange check in this case represents a definite run-time error. Therefore, besides the path-related orange check, the analysis also indicates the error through a red **Non-terminating call** error on `path(10)`.

Bounded Input Values

Checks that are related to bounded inputs fall under the purview of critical orange checks. On the **Results List** pane, for these checks, the **Information** column contains the entry, **Origin: Bounded inputs**. These checks are critical because the software has identified that a run-time error might occur for some program input that is externally bounded. Because the input values are explicitly bounded, the software did not need to make an assumption about the input. Therefore, the input values and the consequent run-time errors are likely to occur in practice.

Most input values can be bounded by external constraints (also known as Data Range Specifications or DRS). See also “Inputs and Stubbing”.

The following example shows an orange check related to bounded input values.

```
int tab[10];
extern int val;
// You specify that val is in [5..10]

void assignElement(int index) {
   int result;
   result = tab[index];
   // Orange Out of bounds array index
 }
void main(void) {
   assignElement(val);
}
```

If you specify a **PERMANENT** data range of 5 to 10 on the variable `val`, verification generates an orange **Out of bounds array index** check on `tab[index]`. The **Result Details** pane provides information about the potential error and states that the check might be related to bounded input values:

```
Warning: array index may be outside bounds: [0..9]
This check may be an issue related to bounded input values
Verifying DRS on extern variable val may remove this orange.
  array size: 10
  array index value: [5 .. 10]
```

Unbounded Input Values

Checks that are related to unbounded inputs do not fall under the purview of critical orange checks. On the **Results List** pane, for these checks, the **Information** column contains the entry, **Origin: Possibly impacted by inputs**. These checks are not considered critical because the software has identified that a run-time error might occur for some program input that is not bounded. Because the input values are not bounded, the software assumes a broad range of input values based on the input data types. The input values and the consequent run-time errors might not occur in practice.

The following example shows an orange check related to unbounded input values that might be identified as a potential run-time error:

```
int tab[10];
extern int val;

void assignElement(int index) {
   int result;
   result = tab[index];
   // Orange Out of bounds array index
 }
void main(void) {
   assignElement(val);
}
```

The verification generates an orange **Out of bounds array index** check on `tab[index]`. The **Result Details** pane provides information about the potential error and states that the check might be related to unbounded input values:

```
Warning: array index may be outside bounds: [0..9]
This check may be an issue related to unbounded input values
If appropriate, applying DRS to extern variable val may remove this orange.
 array size: 10
 array index value: [-2^31 .. 2^31-1]
```

Limit Display of Orange Checks in Polyspace Desktop User Interface

This example shows how to control the number and type of orange checks displayed in the Polyspace desktop user interface. For an equivalent workflow in the Polyspace Access web interface, see “Create Custom Filter Groups in Polyspace Access Web Interface” on page 28-13.

This example shows how to control the number and type of orange checks displayed on the **Results List** pane in the Polyspace desktop user interface. Use the drop-down list in the left of the **Results List** pane toolbar. To reduce your review effort, you can do one of the following:

- Display only the critical orange checks.

  Use the option **Critical checks** in the drop-down list. For more information, see “Critical Orange Checks in Polyspace Code Prover” on page 33-9.
- Limit the number or suppress orange checks for certain check types, using additional options on drop-down list.

  You can add predefined options to the list or create your own options. If you create your own options, you can share the option files to help developers in your organization review at least a certain number or percentage of orange checks.

1. Select **Tools** > **Preferences**.
2. On the **Review Scope** tab, do one of the following:

   - To add predefined options to the drop-down list on the **Results List** pane, select **Include Quality Objectives Scopes**.

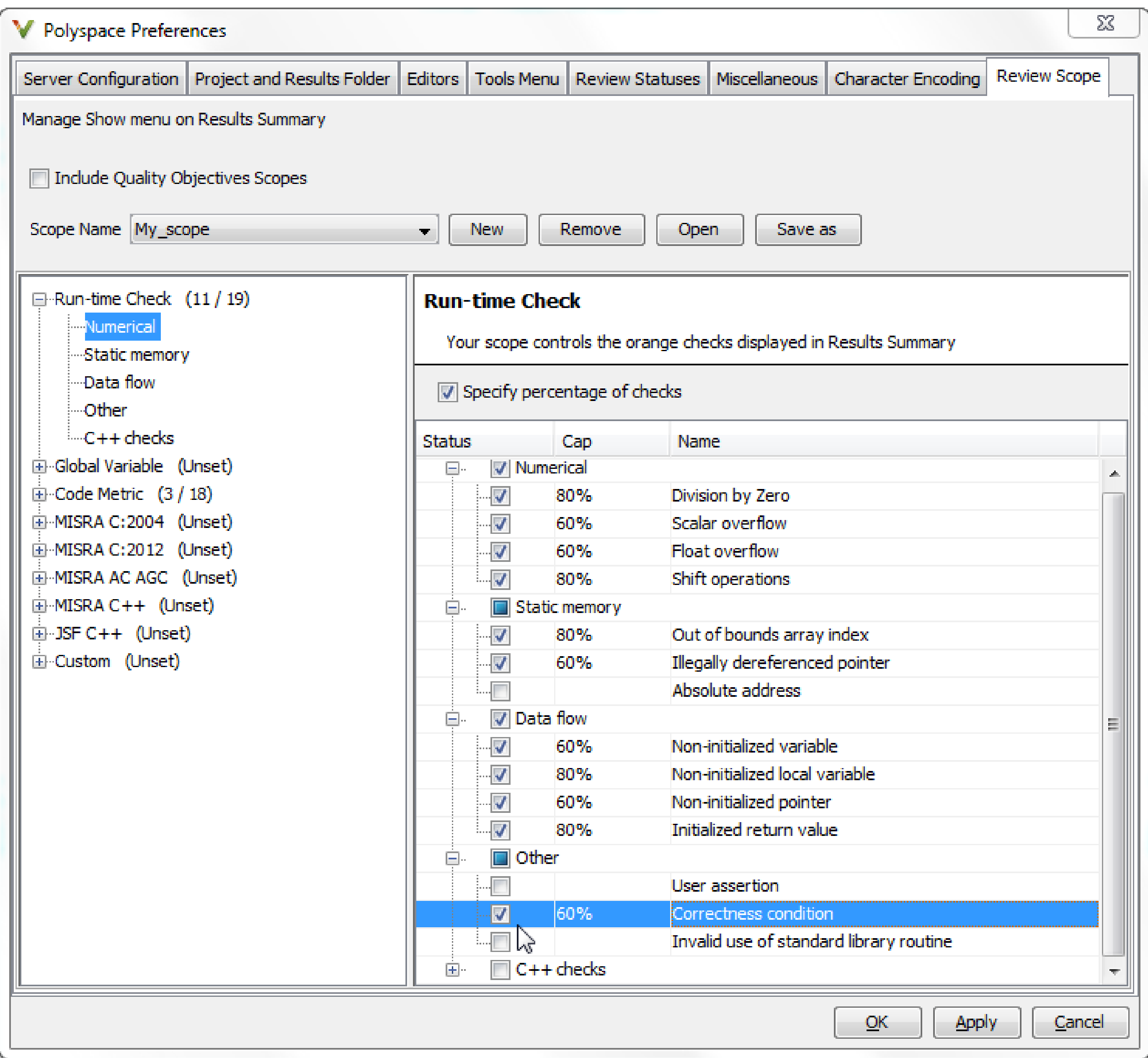
     The **Scope Name** list shows additional options, `HIS`, `SQO-4`, `SQO-5` and `SQO-6`. Select an option to see the limit values.

     In addition to orange checks, the options impose limits on the display of code metrics and coding rule violations. The option `HIS` displays code metrics only. For a detailed explanation of the predefined options, see “Evaluate Polyspace Code Prover Results Against Software Quality Objectives” on page 31-2.
   - To create your own options in the drop-down list on the **Results List** pane, select **New**. Save your option file.

     On the left pane, select **Run-time Check**. On the right pane, to suppress a check completely, clear the box next to the check. To suppress a check partly, specify a percentage less than 100 to display.

     To suppress all checks belonging to a category such as **Numerical**, clear the box next to the category name. For more information on the categories, see “Run-Time Checks”. If only a fraction of checks in a category are selected, the check box next to the category name displays a symbol.

     Instead of a percentage, you can specify a number or the string `ALL`. To specify a number, clear the box **Specify percentage of checks**.



3 Select **Apply** or **OK**.

On the **Results List** pane, the drop-down list on the **Results List** pane displays the additional options.

4 Select the option corresponding to the limits that you want. Only the number or percentage that you specify remain on the **Results List** pane.

- If you specify an absolute number, Polyspace displays that number of orange checks.
- If you specify a percentage, Polyspace calculates that percentage of the total green and orange checks. The software then considers whether green checks alone make up the percentage. If they do not make up the percentage, the software then displays sufficient orange checks to make up the percentage. For example, if you specify 60, the software checks

if 60% of your green and orange checks comprise of green checks only. Otherwise, it displays sufficient orange checks to make up the 60%.

You can use a review scope with percentage specifications to ensure that at least 60% of (green + orange) checks are either green or justified orange. To justify a check, you must assign a **Status** of either `No action planned` or `Justified`. For more information, see “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” on page 23-2.

See Also

Related Examples

- “Filter and Group Results in Polyspace Desktop User Interface” on page 24-2
- “Reduce Orange Checks in Polyspace Code Prover” on page 33-17
- “Critical Orange Checks in Polyspace Code Prover” on page 33-9

Reduce Orange Checks in Polyspace Code Prover

An orange check indicates that Polyspace detects a possible run-time error but cannot prove it. To help Polyspace produce more proven results, you can:

- Specify appropriate verification options.
- Follow appropriate coding practices.

You can also limit the number and family of orange checks displayed on **Results List**. For more information, see “Limit Display of Orange Checks in Polyspace Desktop User Interface” on page 33-14.

You can take one or more of the following actions for orange check reduction.

Provide Context for Verification

This example shows how to provide additional information about run-time execution of your code. Sometimes, the code you provide does not contain this information. For instance:

- You do not have a `main` function
- You have a function that is declared but not defined.
- You have function arguments whose values are available only at run-time.
- You have concurrently running functions that are intended for execution in a specific sequence.

Without sufficient information, Polyspace Code Prover cannot verify the presence or absence of run-time errors.

Constrain Orange Sources

You can often address the bulk of orange checks by constraining relatively fewer number of orange sources.

In your verification results, you can see a list of sources such as volatile variables and stubbed functions that can cause multiple orange checks. To see this list, click the button on the **Result Details** pane. Constrain these sources so that you can remove most orange checks from overapproximation. In the longer term, instead of reacting to orange sources during review and constraining them for a later run, devise an efficient strategy for constraining likely orange sources during the first run itself.

See “Sources of Orange Checks” for types of orange sources and how to constrain them.

Commonly Used Context Specifications

To provide more context for verification and reduce orange checks, use these methods.

Method	Example
Define how the `main` generated by Polyspace initializes variables and calls functions	“Code Prover Verification”
Define ranges for global variables and function arguments.	“Create Constraint Template from Code Prover Analysis Results” on page 14-3

Method	Example
Define execution sequence for multitasking code.	"Configuring Polyspace Multitasking Analysis Manually" on page 15-17
Map an imprecisely analyzed function to a standard function for precise results at the function call sites.	`-code-behavior-specifications`

Improve Verification Precision

This example shows how to improve the precision of your verification. Increased precision reduces orange checks, but increases verification time.

Use the following options. In the Polyspace user interface, the options appear on the **Configuration** pane under the **Precision** node.

Option	Purpose
`Precision level (-O0 | -O1 | -O2 | -O3)`	Specify the verification algorithm.
`Verification level (-to)`	Specify the number of times the Polyspace verification process runs on your source code.
`Improve precision of interprocedural analysis (-path-sensitivity-delta)`	Propagate greater information about function arguments into the called function.
`Sensitivity context (-context-sensitivity)`	If a function contains a red and green check on the same instruction from two different calls, display both checks instead of an orange check.

Follow Coding Rules

This example shows how to follow coding rules that help Polyspace Code Prover prove the presence or absence of run-time errors. If your code follows certain subsets of MISRA coding rules, Polyspace can verify the presence or absence of run-time errors more easily.

1. Check whether your code follows the relevant subset of coding rules. For instance:

   a. In the user interface of the Polyspace desktop products, on the **Configuration** pane, depending on the code, select one of the options under the **Coding Rules** node.

Type of Code	Option	Rule Description
Handwritten C code	**Check MISRA C:2004** or **Check MISRA C:2012**	• "Software Quality Objective Subsets (C:2004)" on page 16-11 • "Software Quality Objective Subsets (C:2012)" on page 16-19

Type of Code	Option	Rule Description
Generated C code	**Check MISRA AC AGC**	"Software Quality Objective Subsets (AC AGC)" on page 16-15
Handwritten C++ code	**Check MISRA C++ rules**	"Software Quality Objective Subsets (C++)" on page 16-23

b From the option drop-down list, select `SQO-subset1` or `SQO-subset2`.

2. Run verification and review your results.
3. Fix the coding rule violations.

Reduce Application Size

Sometimes, the application size causes a loss of precision.

In a relatively smaller application, Code Prover retains more precise information about variable ranges. For instance, suppose a variable takes these values: {-2,-1,2,10,15,16,17}. If this variable is the denominator in a division, Code Prover shows a green **Division by zero** as long as it retains this precise information. If the application size grows beyond a certain point, to reduce computational complexity, Code Prover approximates this range to, for instance, {-2,2} U {10} U {15,17}. Now, if the variable is used for division, Code Prover shows an orange **Division by zero**.

To improve precision, you can divide the application into multiple modules. Verify each module independently of the other modules. You can review the complete results for one module, while the verification of the other modules are still running.

- You can let the software modularize your application. The software divides your source files into multiple modules such that the interdependence between the modules is as little as possible. To begin, select **Tools** > **Run Modularize**.
- If you are running verification in the user interface, you can create multiple modules in your project and copy source files into those modules.
- You can perform a file-by-file verification. Each file constitutes a module by itself. See `Verify files independently (-unit-by-unit)`.

Follow Verification Setup Suggestions

In some cases, Polyspace Code Prover can provide suggestions for better verification setup. At the end of each verification, Code Prover runs an advisor on the verification results. The advisor queries the results and looks for issues that can increase verification time and reduce precision. If the advisor can find issues (based on built-in rules), it suggests analysis options that can be used to work around the issues in future runs.

For instance:

- If you are generating a `main` that calls too many functions already called elsewhere, you might see a suggestion to reduce the number of called functions.
- If most orange checks are coming from global variables, you might see a suggestion to add constraints on those global variables.

You see these suggestions at the end of the verification log. The suggestions look like the following:

```
**********************************************************
***
*** Beginning Advisor
***
**********************************************************
Suggestion (polyspace.advisor.MainGeneratorCustomNeeded):
 Issue: The generated main calls 54 functions, which might be too many and lead to loss of perfor
 Cause: 49 functions are called both by the generated main and pointers to functions. These funct
 Fix: Use -main-generator-calls custom=__bswap_16,__bswap_32,__bswap_64,apply_threshold,get_sum


**********************************************************
***
*** Advisor done
***
**********************************************************
```

If the fix suggestion mentions a Polyspace analysis option, you can copy the option and paste into your Polyspace options file on page 12-5. Even when the fix suggestions do not involve options that can be directly copied, you can use the suggestions as starting points for orange check reduction.

See Also

More About

- “Orange Checks in Polyspace Code Prover” on page 33-2
- “Managing Orange Checks in Polyspace Code Prover” on page 33-5

Troubleshooting

Troubleshooting in Polyspace Code Prover

- “View Error Information When Analysis Stops” on page 34-3
- “Troubleshoot Compilation and Linking Errors” on page 34-6
- “Reduce Memory Usage and Time Taken by Polyspace Analysis” on page 34-10
- “Identify Root Causes of Code Prover Red or Orange Checks” on page 34-15
- “Contact Technical Support About Issues with Running Polyspace” on page 34-18
- “Fix Error: Polyspace Cannot Find Server” on page 34-21
- “Fix Error: Job Manager Cannot Write to Database” on page 34-22
- “Resolve Error: No Compilation Unit Detected in Your Build” on page 34-23
- “Create Polyspace Projects from Build Systems That Use Unsupported Compilers” on page 34-25
- “Fix Slow Build Process When Polyspace Traces Build” on page 34-31
- “Check if Polyspace Supports Build Scripts” on page 34-32
- “Troubleshoot Project Creation from MinGW Build” on page 34-33
- “Troubleshoot Project Creation from Visual Studio Build” on page 34-34
- “Resolve polyspace-autosar Error: Could Not Find Include File” on page 34-35
- “Resolve polyspace-autosar Error: Conflicting Universal Unique Identifiers (UUIDs)” on page 34-37
- “Resolve polyspace-autosar Error: Data Type Not Recognized” on page 34-38
- “Fix Polyspace Compilation Errors About Undefined Identifiers” on page 34-40
- “Fix Polyspace Compilation Errors About Unknown Function Prototype” on page 34-43
- “Fix Polyspace Compilation Errors Related to #error Directive” on page 34-44
- “Fix Polyspace Compilation Errors About Large Objects” on page 34-45
- “Fix Polyspace Compilation Errors Related to Generic Compiler” on page 34-47
- “Fix Polyspace Compilation Errors Related to GNU Compiler” on page 34-48
- “Fix Polyspace Compilation Errors Related to Visual Compilers” on page 34-49
- “Fix Polyspace Compilation Errors Related to Keil or IAR Compiler” on page 34-51
- “Fix Polyspace Compilation Errors Related to Diab Compiler” on page 34-52
- “Fix Polyspace Compilation Errors Related to Green Hills Compiler” on page 34-54
- “Fix Polyspace Compilation Errors Related to TASKING Compiler” on page 34-56
- “Fix Polyspace Compilation Errors Related to Texas Instruments Compilers” on page 34-58
- “Fix Polyspace Compilation Errors About In-Class Initialization (C++)” on page 34-59
- “Fix Polyspace Linking Errors About Conflicting Declarations in Different Translation Units” on page 34-60
- “Fix Errors from Use of Polyspace Header Files” on page 34-65
- “Fix Polyspace Linking Errors Related to extern "C" Blocks” on page 34-67

- “Fix Polyspace Compilation Errors About Namespace std Without Prefix” on page 34-69
- “Fix Polyspace Compilation Warnings Related to Assertion or Memory Allocation Functions” on page 34-70
- “Troubleshoot Java Incompatibility in Polyspace Plugin for Eclipse” on page 34-71
- “Fix Issues When when Integrating Polyspace with MATLAB and Simulink” on page 34-73
- “Check Why Polyspace Functions are Unavailable in MATLAB” on page 34-75
- “Fix MATLAB Crashes Referring to Polyspace in matlabroot” on page 34-76
- “Reasons for Unchecked Code” on page 34-77
- “Identify Why Some Files or Functions are Missing in Polyspace Results” on page 34-83
- “Fix Polyspace Overapproximations on Standard Library Math Functions” on page 34-86
- “Avoid Red Checks in Unreachable Code When Using C++ STL Containers” on page 34-87
- “Fix Insufficient Memory Errors During Polyspace Report Generation” on page 34-88
- “Fix Polyspace Errors Related to Temporary Files” on page 34-91
- “Fix Errors or Slow Polyspace Runs from Disk Defragmentation and Anti-virus Software” on page 34-93
- “Fix SQLite I/O Errors on Running Polyspace” on page 34-95
- “Fix License Error –4,0 When Running Polyspace” on page 34-96
- “Fix Errors Applying Custom Annotation Format for Polyspace Results” on page 34-97

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 `#include`-d in your code. The issue does not block the analysis by itself, but if the include file contains the definition of a variable that you use in your source code, you can face an error later.
	Additional information about the analysis.

2 To diagnose and fix each error, you can do the following:

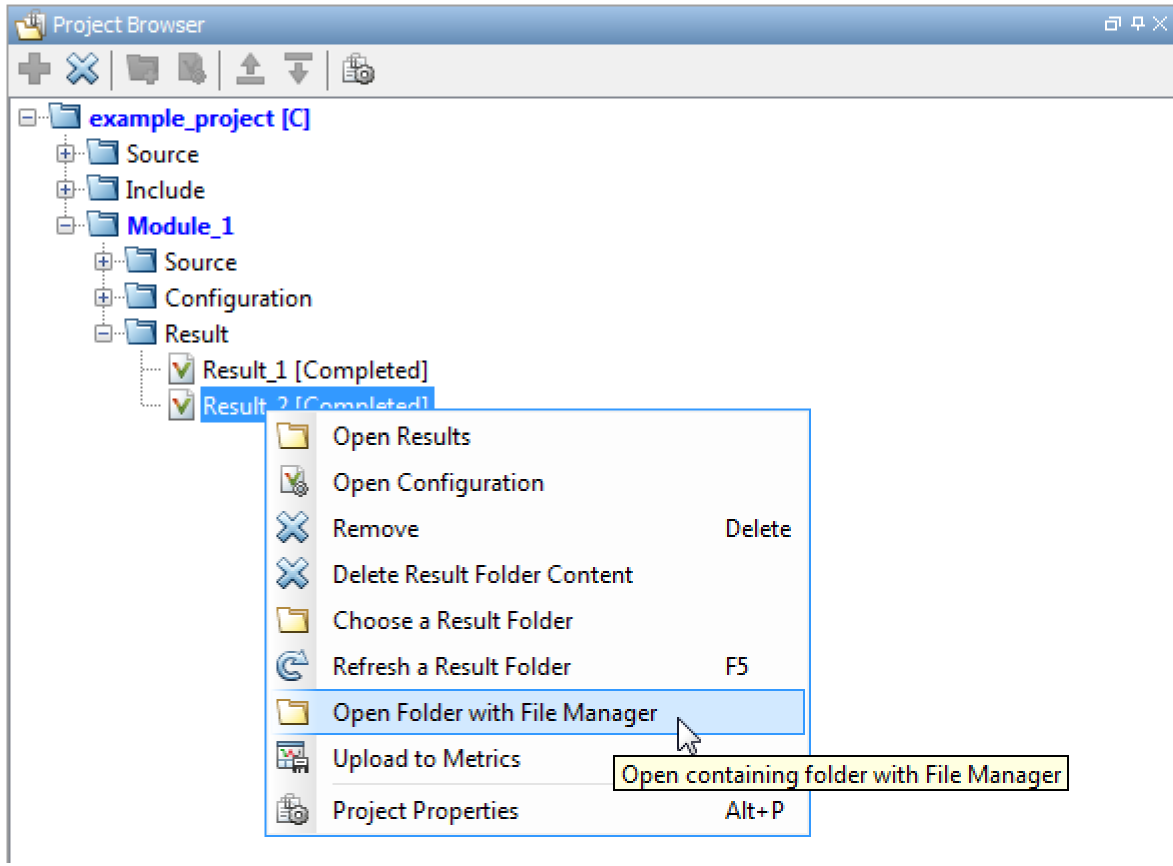
   - To see further details about the error, select the error message. The details appear in a **Detail** window below the **Output Summary** tab.
   - To open the source code at the line containing the error, double-click the message.

Tip To search the error messages for a specific term, on the **Search** pane, enter your search term. From the drop down list on this pane, select **Output Summary** or **Run Log**. If the **Search** pane is not open by default, select **Windows > Show/Hide View > Search**.

View Error Information in Log File

You can view errors directly in the log file. The log file is in your results folder. To open the log file:

1 Right-click the result folder name on the **Project Browser** pane. From the context menu, select **Open Folder with File Manager**.



2 Open the log file, `Polyspace_R20##n_ProjectName_date-time.log`

3 To view the errors, scroll through the log file, starting at the end and working backward.

The following example shows sample log file information. The error has occurred because the C++ option `-class-analyzer custom=arg` was used, but the analysis cannot find *arg* in the source code.

```
----------------------------------------------------------------------
User Program Error: Argument of option -class-analyzer not found.
|                   Class or typedef MyClass does not exist.
|Please correct the program and restart the verifier.
----------------------------------------------------------------------
-------------------------------------------------------------------
---                                                             ---
---  Verifier has encountered an internal error.                ---
---  Please contact your technical support.                     ---
---                                                             ---
-------------------------------------------------------------------
Failure at: Sep 24, 2009 17:16:26
User time for polyspace-code-prover: 25.6real, 25.6u + 0s
                                                           (0gc)
Error: Exiting because of previous error
***
*** End of Polyspace Verifier analysis
***
```

See Also

`Stop analysis if a file does not compile (-stop-if-compile-error)` | `File does not compile`

Troubleshoot Compilation and Linking Errors

Run Polyspace verification on code that builds successfully with your compiler. Once your code builds successfully, set up a Polyspace project in one of these ways:

- Trace your build system.

  The software creates a project from your build scripts. It sets appropriate Polyspace analysis options to emulate your build options.
- If you cannot trace your build system, create a Polyspace project manually.

  Add your sources and includes to the project. Change the default analysis options, if required.

For more information, see “Configure and Run Analysis”.

The following issue occurs more often if you manually set up your project.

Issue

Before verification and detection of run-time errors, Polyspace compiles your code and detects compilation and linking errors. Even if your code builds successfully with your compiler, you still get compilation errors with Polyspace.

Verification running

Compile: 88% Elapsed time: 00:00:08

Total: 14% Total elapsed time: 00:00:08

Type	Message	File	Line	Col
(i)	C verification starts at Mon Dec 07 16:48:05 2015			
(i)	6 core(s) detected but the verification uses 4 core(s).			

Compilation Phase

Type	Message	File	Line	Col
(i)	C verification starts at Thu Dec 17 22:26:17 2015			
(i)	6 core(s) detected but the verification uses 4 core(s).			
(x)	identifier "x" is undefined	my_file.c	1	
(!)	Failed compilation.	my_file.c		
(x)	Verifier has detected compilation error(s) in the code.			
(x)	Exiting because of previous error			

Compilation Failure

Possible Cause: Deviations from Standard

The Polyspace compiler strictly follows a C or C++ standard. See `C standard version (-c-version)` and `C++ standard version (-cpp-version)`. If your compiler allows deviation from the Standard, the Polyspace compilation that uses default options cannot emulate your compiler. For

instance, your compiler can allow certain non-ANSI keywords that Polyspace does not recognize by default.

To guarantee absence of certain run-time errors, the default Polyspace compilation strictly follows the standard. Specific compilers allow specific deviations from this standard and follow internal algorithms to compile your code. Without explicit knowledge of your compiler behavior, Polyspace cannot accommodate those deviations. Accommodating these deviations through some arbitrary internal algorithms can compromise the final analysis results, if the Polyspace algorithm does not match your compiler's algorithm.

Check the error message that caused the compilation failure and see if you can identify some deviation from the standard. The error message shows the line number that caused the compilation failure. If you run verification from the user interface, you can click the error message and navigate to the corresponding line of code.

Solution

Change analysis options to emulate your compiler more closely by manually adjusting these options:

Option	Purpose
"Target and Compiler" options	Using these predefined options, you can specify your compiler behavior directly and work around known deviations from the standard. Often, setting `Compiler (-compiler)` appropriately is enough to emulate your compiler.
• `Preprocessor definitions (-D)` • `Command/script to apply to preprocessed files (-post-preprocessing-command)`	Using these options, you can sometimes work around unknown deviations from the standard. For instance, you can use these options to replace unrecognized keywords from your preprocessed code with closely matching recognized keywords, or remove them completely. Because you do not change your source code, the options allow you to work around compilation errors while keeping your source code intact.

See also "View Error Information When Analysis Stops" on page 34-3.

For specific types of compilation errors, see "Troubleshoot Compilation Errors".

If you cannot solve your compilation error, contact MathWorks Technical Support and provide your compiler name for better support. See "Contact Technical Support About Issues with Running Polyspace" on page 34-18.

Possible Cause: Linking Errors

Even if a single compilation unit compiles successfully, you get a linking error because of mismatch between two compilation units. For instance, you define the same function in two `.c` files with different argument or return types.

Common compilation toolchains do not store information about function prototypes during the linking process. Therefore, despite these types of linking errors, the build does not fail. To guarantee absence of certain run-time errors, Polyspace does not continue analysis when such linking errors occur.

Solution

Fix the linking errors that Polyspace detects. Even if your build process allows these errors, you can have unexpected results during run time. For instance, if two function definitions with the same name but conflicting prototypes exist in your code, when you call the function, the result can be unexpected.

When a linking error occurs, the error message shows the location in your file where Polyspace compilation fails. Previous warning messages show the location of the conflicts that lead to the linking error. Using the line numbers in those messages (or by clicking the messages if you run analysis from the user interface), you can navigate to the location of the conflicts in your code.

For instance, in these messages, compilation fails because of conflicting function return types. The failure occurs on line 5 in `file2.c` when the function is called. The previous warning messages for line 1 in `file1.c` and line 1 in `file2.c` show the locations where the conflicts occur.

Type	Message	File	Line	Col
	C verification starts at Thu Dec 17 22:01:26 2015			
	6 core(s) detected but the verification uses 4 core(s).			
	global declaration of 'f' function has incompatible type with its defi...	file2.c	1	
	other location for previous warning	file1.c	1	
	calling function `f' with incompatible return type	file2.c	5	

Detail

```
File myFile.c                                                  line 1

Warning: global declaration of 'f' function has incompatible type with its definition
 Declared function type has incompatible return type with definition.
 Declared 'int' (size 32) type incompatible with defined 'float' (size 32) type.
 Definition: function with return type float
 Declaration: function with return type int
```

For specific types of linking errors, see "Troubleshoot Compilation Errors".

Possible Cause: Conflicts with Polyspace Function Stubs

Polyspace uses its own implementation of standard library functions for more efficient verification. If your compiler redeclares and redefines a standard library function, you can get a warning or error when you invoke the function.

The error implies that Polyspace found the redeclaration but cannot find the body of your redefined library function. The verification continues to use the Polyspace implementation of the function but provides a warning. If your redefined function has a different signature from the normal signature of the function, the verification stops with an error.

Warnings and errors of this type often refer to the file `__polyspace__stdstubs.c`. This file contains prototypes for the Polyspace implementation of standard library functions. The file is located in *`polyspaceroot`*`\polyspace\verifier\cxx\polyspace_stubs\`. *`polyspaceroot`* is the Polyspace installation folder.

Solution

If you know the location of the file that contains the body of your redefined standard library function, add the file to your verification. For more information, see “Fix Errors from Use of Polyspace Header Files” on page 34-65.

If you do not have the function body available:

- If you see a warning of this type, you can ignore the warning. The verification results are based on Polyspace implementations of standard library functions. If your compiler redefinition closely matches the standard library function specifications, the verification results are still applicable for code compiled with your compiler.
- If you see an error:

  1. Define the macro `__polyspace_no_`*`function_name`* in your project. For instance, if an error occurs because of a conflict with the definition of the `sprintf` function, define the macro `__polyspace_no_sprintf`. For information on how to define macros, see `Preprocessor definitions (-D)`.

     The macro disables the use of Polyspace implementations of the standard library function. The software stubs the standard library function like any other undefined function. You do not have an error because of signature mismatch with the Polyspace implementations.
  2. Contact MathWorks Technical Support and provide information about your compiler.

For some standard library functions, such as `assert`, and memory allocation functions such as `malloc` and `calloc`, Polyspace continues to use its own implementations, even if you redefine the function and provide the function body. For more information, see “Fix Polyspace Compilation Warnings Related to Assertion or Memory Allocation Functions” on page 34-70.

Reduce Memory Usage and Time Taken by Polyspace Analysis

In this section...

"Issue" on page 34-10

"Possible Cause: Temporary Folder on Network Drive" on page 34-10

"Possible Cause: Anti-Virus Software" on page 34-10

"Possible Cause: Large and Complex Application" on page 34-11

"Possible Cause: Too Many Entry Points for Multitasking Applications" on page 34-13

Issue

The verification is stuck at a certain point for a long time. Sometimes, after the period of inactivity exceeds an internal threshold, the verification stops or you get an error message:

```
The analysis has been stopped by timeout.
```

For large projects with several hundreds of source files or millions of lines of code, you might run into the same issue in another way. The verification stops with the error message:

```
Fatal error: Not enough memory
```

If you have a multicore system with more than four processors, try increasing the number of processors by using the option `-max-processes`. By default, the verification uses up to four processors. If you have fewer than four processors, the verification uses the maximum available number. You must have at least 4 GB of RAM per processor for analysis. For instance, if your machine has 16 GB of RAM, do not use this option to specify more than four processors.

If the verification still takes too long, to improve the speed and make the verification faster, try one of the solutions below.

Possible Cause: Temporary Folder on Network Drive

Polyspace produces some temporary files during analysis. If the folder used to store these files is on a network drive, the analysis can slow down.

Solution: Change Temporary Folder

Change your temporary folder to a path on a local drive.

To learn how Polyspace determines the temporary folder location, see "Storage of Temporary Files During Polyspace Analysis" on page 3-6.

Possible Cause: Anti-Virus Software

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.

Configure Exceptions for Polyspace Processes

Check the processes running and see if an anti-virus is consuming large amount of memory.

See “Fix Errors or Slow Polyspace Runs from Disk Defragmentation and Anti-virus Software” on page 34-93.

Possible Cause: Large and Complex Application

The verification time depends on the size and complexity of your code.

If the application contains greater than 100,000 lines of code, the verification can sometimes take a long time. Even for smaller applications, the verification can take long if it involves complexities such as structures with many levels of nesting or several levels of aliasing through pointers. You can see the number of lines excluding comments towards the beginning of the analysis log in your results folder. Search for the string:

```
Number of lines without comments
```

However, if verification with the default options takes unreasonably long or stops altogether, there are multiple strategies to reduce the verification time. Each strategy involves reducing the complexity of verification in some way.

Solution: Use Polyspace Bug Finder First

Use Polyspace Bug Finder first to find defects in your code. Some defects that Polyspace Bug Finder finds can translate to a red error in Polyspace Code Prover. Once you fix these defects, use Polyspace Code Prover for a more rigorous verification.

Solution: Modularize Application

You can divide the application into multiple modules. Verify each module independently of the other modules. You can review the complete results for one module, while the verification of the other modules are still running.

- You can let the software modularize your application. The software divides your source files into multiple modules such that the interdependence between the modules is as little as possible.
  - For automatic modularization from the Polyspace user interface, see “Modularize Large Projects in Polyspace Desktop User Interface” on page 2-20.
  - For automatic modularization from the command line, see “Modularize Polyspace Analysis at Command Line Based on an Initial Interdependency Analysis” on page 4-15.
- If you are running verification in the user interface, you can create multiple modules in your project and copy source files into those modules. To begin, right click a project and select **Create new module**.
- You can perform a file-by-file verification. Each file constitutes a module by itself. See `Verify files independently (-unit-by-unit)`.

When you divide your complete application into modules, each module has some information missing. For instance, one module can contain a call to a function that is defined in another module. The software makes certain assumptions about the undefined functions. If the assumptions are broader than an actual representation of the function, you see an increase in orange checks from overapproximation. For instance, an error management function might return an `int` value that is either 0 or 1. However, when Polyspace cannot find the function definition, it assumes that the function returns all possible values allowed for an `int` variable. You can narrow down the assumptions by specifying external constraints.

When modularizing an application manually, you can follow your own modularization approach. For instance, you can copy only the critical files that you are concerned about into one module, and verify them. You can represent the remaining files through external constraints, provided you are confident that the constraints represent the missing code faithfully. For instance, the constraints on an undefined function represent the function faithfully if they represent the function return value and also reproduce other relevant side effects of the function. To specify external constraints, use the option `Constraint setup (-data-range-specifications)`.

Solution: Choose Lower Precision Level or Verification Level

If your verification takes too long, use a lower precision level or a lower verification level. Fix the red errors found at that level and rerun verification.

- The precision level determines the algorithm used for verification. Higher precision leads to greater number of proven results but also requires more verification time. For more information, see `Precision level (-O0 | -O1 | -O2 | -O3)`.
- The verification level determines the number of times Polyspace runs on your source code. For more information, see `Verification level (-to)`.

The verification results from lower precision can contain more orange checks. An orange check indicates that the analysis considers an operation suspect but cannot prove the presence or absence of a run-time error. You have to review an orange check thoroughly to determine if you can retain the operation. By increasing the number of orange checks, you are effectively increasing the time you spend reviewing the verification results. Therefore, use these strategies only if the verification is taking too long.

Solution: Reduce Code Complexity

Both for better readability of your code and for shorter verification time, you can reduce the complexity of your code. Polyspace calculates code complexity metrics from your application, and allows you to limit those metrics below predefined values.

For more information, see:

- “Code Metrics”: List of code complexity metrics and their recommended upper limits
- “Compute Code Complexity Metrics Using Polyspace” on page 16-47: How to set limits on code complexity metrics

Solution: Compute Global Variable Sharing and Usage Only

Run a less extensive Code Prover analysis on your application that computes global variable sharing and usage only. You can then run a full analysis component-by-component for run-time error detection. See `Show global variable sharing and usage only (-shared-variables-mode)`.

Solution: Enable Approximations

Depending on your situation, you can choose scaling options to enable certain approximations. Often, warning messages indicate that you must use those options to reduce verification.

Situation	Option
Your code contains structures that are many levels deep.	`Depth of verification inside structures (-k-limiting)`

Situation	Option
The verification log contains suggestions to inline certain functions.	`Inline (-inline)`

Solution: Remove Parts of Code

You can try to remove code from third-party libraries. The software uses stubs for functions that are not defined in files specified for the Polyspace analysis.

Although the analysis time is reduced, you can see an increase in orange checks because of Polyspace assumptions about stubbed functions. See "Code Prover Assumptions About Stubbed Functions".

Possible Cause: Too Many Entry Points for Multitasking Applications

If your code is intended for multitasking and you provide many Tasks, verification can take a long time. The following warning can appear:

```
Warning: Polyspace detected that the application has x tasks and y critical sections.
         The precision of the analysis will be reduced to avoid scaling issues.
         Check if you can reduce the number of tasks and critical sections in the configuration.
```

If you receive this warning, it means that Polyspace is switching to a less precise analysis mode to complete the verification in a reasonable amount of time. In this less precise mode, the verification can consider some shared variables as full-range and cause orange checks from overapproximation.

Solution

See if you can reduce the number of entry points that you specify. If you know that some of your entry point functions do not execute concurrently, you do not have to specify them as separate entry points. You can call those functions sequentially in a wrapper function, and then specify the wrapper function as your entry point.

For instance, if you know that the entry point functions `task1`, `task2`, and `task3` do not execute concurrently:

1. Define a wrapper function `task` that calls `task1`, `task2`, and `task3` in all possible sequences.

```
void task() {
   volatile int random = 0;
   if (random) {
       task1();
       task2();
       task3();
  } else if (random) {
       task1();
       task3();
       task2();
  } else if (random) {
       task2();
       task1();
       task3();
  } else if (random) {
       task2();
       task3();
       task1();
```

```
    } else if (random) {
        task3();
        task1();
        task2();
    } else {
        task3();
        task2();
        task1();
    }
}
```

2 Instead of `task1`, `task2`, and `task3`, specify `task` for the option `Tasks (-entry-points)`.

For an example of using a wrapper function as an entry point, see "Configuring Polyspace Multitasking Analysis Manually" on page 15-17.

See Also

More About

- "Modularize Polyspace Analysis by Using Build Command" on page 4-5
- "Run Polyspace on AUTOSAR Code" on page 8-14
- "Modularize Large Projects in Polyspace Desktop User Interface" on page 2-20
- "Modularize Polyspace Analysis at Command Line Based on an Initial Interdependency Analysis" on page 4-15

External Websites

- Resolving Scaling Problems in Code Prover

Identify Root Causes of Code Prover Red or Orange Checks

Issue

After verification, Polyspace Code Prover highlights operations in your code with specific colors depending on whether the operation can cause a run-time error. See “Code Prover Result and Source Code Colors” on page 32-2.

It is not immediately clear why the verification highlights a specific operation in red (definite run-time error) or orange (potential run-time error). Even if you understand the cause of an error, it is not immediately clear where to fix it.

Possible Cause: Relation to Prior Code Operations

Often a run-time error in a specific operation is related to prior operations in your code.

For instance, an operation overflows because of a large operand value but the operand acquires that value in previous operations.

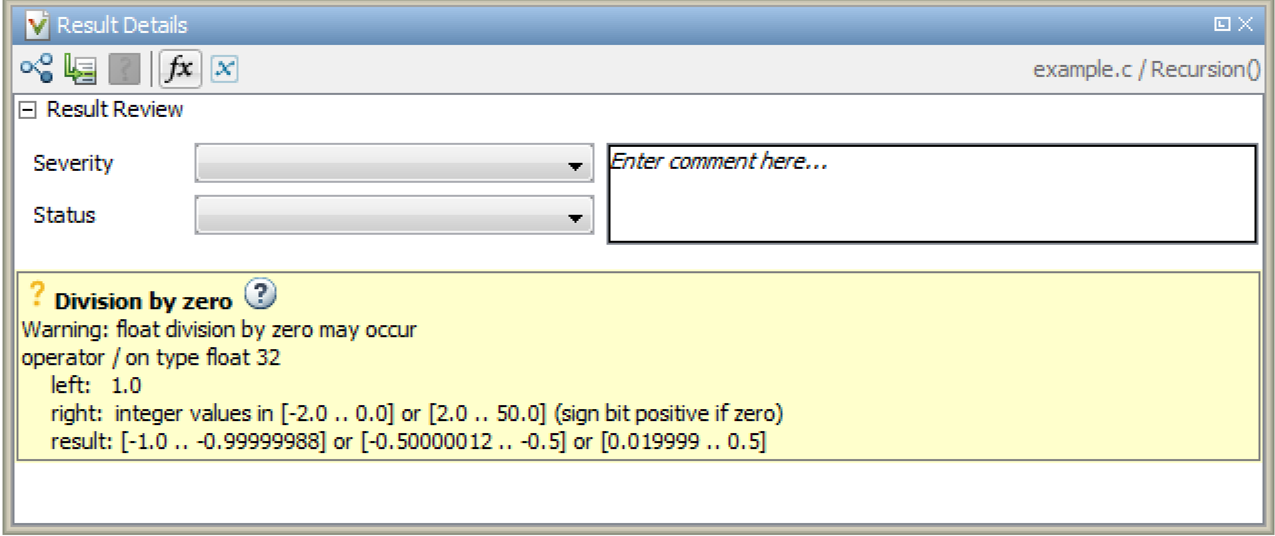
Solution

To investigate how a prior operation triggers a run-time error in the current operation, do the following:

- View the message associated with the verification result on the current operation.

  The message appears in the **Result Details** pane or in tooltips on the operation in the **Source** pane. The message shows you how to investigate the result further.

  For instance, the message below shows that the right operand can be zero. To determine how the operand variable acquires the value zero, you have to browse through previous operations that write to the variable.



- Browse prior operations in your code that are related to the current operation.

The Polyspace user interface provides features for easy navigation between specific points in your code. For instance, you can navigate from a function name to the function definition.

Identify a suitable place in your code where you can implement the fix.

For specific information on how to review each check type, see “Reviewing Code Prover Run-Time Checks” on page 32-7.

Possible Cause: Software Assumptions

If you do not provide your complete application or the external information required for verification, the software has to make certain assumptions about the missing code or external information.

For instance, if you do not provide a `main` function, the software generates a `main` that calls only the uncalled functions. If `func1` calls `func2`, the generated `main` does not call `func2` again. The verification checks for run-time errors in `func2` only from the call context in `func1`.

The assumptions are such that they apply to most applications. However, in a few cases, the default assumptions might not describe your run-time environment accurately. If the assumptions are not what you expect, the verification results can be unexpected.

Solution

See if you can trace your verification result to a software assumption. For a partial list of assumptions, see “Code Prover Analysis Assumptions”. An additional list of assumptions is provided in `codeprover_limitations.pdf` in *polyspaceroot*`\polyspace\verifier\code_prover_desktop`.

Often, you can change the default assumptions using certain options.

- “Target and Compiler”: See if you must set an option to emulate your compiler behavior.

  For instance, if you want quotients of division operations to be rounded down instead of rounded up, use the option `Division round down (-div-round-down)`.
- “Inputs and Stubbing”: See if you have to externally constrain some variables.

  For instance, if you want to constrain a global variable within a specific range, use the option `Constraint setup (-data-range-specifications)`.
- “Multitasking”: See if you forgot to specify some tasks or protection mechanisms.

  For instance, if you want to specify that a function represents a nonpreemptable interrupt, use the option `Interrupts (-interrupts)`.
- “Code Prover Verification”: If you are verifying a module without a `main`, see if the generated `main` initializes your global variables and calls your functions in the right order.

  For instance, if you want the generated `main` to call all your functions, use the option `Functions to call (-main-generator-calls)` with argument `all`.
- “Verification Assumptions”: See if the global verification assumptions are appropriate for your run-time environment.

  For instance, if you want the verification to consider that unknown pointers can be `NULL`-valued, use the option `Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)`.

- “Check Behavior”: See if the run-time check specifications are appropriate for your run-time environment.

  For instance, if you want the `Illegally dereferenced pointer` check to allow pointer arithmetic across fields of a structure, use the option `Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)`.

If you still cannot understand your result, contact MathWorks Technical Support for help with interpreting your result. If you cannot share your actual verification results, provide only certain essential information about your result. See “Contact Technical Support About Issues with Running Polyspace” on page 34-18.

Contact Technical Support About Issues with Running Polyspace

To contact MathWorks Technical Support, use this page. You need a MathWorks Account login and password. For faster turnaround with an issue in Polyspace, besides the required system information, provide appropriate code that reproduces the issue or the verification log file.

Provide System Information

When you enter a support request, provide the following system information:

- Hardware configuration
- Operating system
- Polyspace and MATLAB license numbers
- Specific version numbers for Polyspace products
- Installed Bug Report patches

To obtain your configuration information, do one of the following:

- In the Polyspace user interface, select **Help** > **About**.
- At the command line, navigate to your Polyspace installation folder, for instance `C:\Program Files\Polyspace\R2023a` (Windows) or `/usr/local/Polyspace/R2023a` (UNIX), and run the command that corresponds to your product and platform:

Product	Command

Polyspace Bug Finder
 	
Polyspace Code Prover
	• **UNIX**
	`polyspace/bin/polyspace-bug-finder -ver`
	• **Windows**
	`polyspace\bin\polyspace-bug-finder.exe -ver`

Polyspace Bug Finder Server
 	
Polyspace Code Prover Server
	• **UNIX**
	`polyspace/bin/polyspace-bug-finder-server -ver`
	• **Windows**
	`polyspace\bin\polyspace-bug-finder-server.exe -ver`

If you configure Polyspace to offload the analysis from a client machine to a server machine, to obtain the system configuration of the server machine from the client machine, add options `-batch -scheduler` *`MJSName@host`* to the command. For example:

```
polyspace/bin/polyspace-bug-finder -ver -batch -scheduler MJSName@host
```

Here, *`MJSName`* is the name of the MATLAB Job Scheduler on the head node of the MATLAB Parallel Server cluster and *`host`* is the host name of the server machine that hosts the head node of this cluster.

Provide Information About the Issue

Depending on the issue, provide appropriate artifacts to help Technical Support understand and reproduce the issue.

Compilation Errors

If you face compilation issues with your project, see “Troubleshoot Compilation Errors”. If you are still having issues, contact technical support with the following information:

- The analysis log.

  The analysis log is a text file generated in your results folder and titled `Polyspace_`*`version_project_date_time`*`.log`. It contains the error message, the options used for the analysis and other relevant information.
- The source files related to the compilation error or the complete results folder if possible.

  If you cannot provide the source files:

  - Try to provide a screenshot of the source code section that causes the compilation issue.
  - Try to reproduce the issue with a different code. Provide that code to technical support.

Polyspace as You Code writes the contents of compilation error messages to a log file. The log is generated in your results folder and titled `polyspace_err.log`. Provide this log to technical support if you encounter a compilation issue with Polyspace as You Code.

Errors in Project Creation from Build Systems

If you face errors in creating a project from your build system, see “Troubleshoot Project Creation”.

If you are still having issues, contact technical support with debug information. To provide the debug information:

1. Run `polyspace-configure` at the command line with the option `-easy-debug`. For instance:

   ```
   polyspace-configure options -easy-debug pathToFolder buildCommand
   ```

   Here:

   - *`options`* is the list of `polyspace-configure` options that you typically use.
   - *`buildCommand`* is the build command that you use, for instance, `make`.
   - *`pathToFolder`* is the folder where you want to store debug information, for instance, `C:\Temp\BuildLogs`. After a `polyspace-configure` run, the path provided contains a zipped file ending with `pscfg-output.zip`. The zipped file contains debug information only and does not contain source files traced in the build.

   Make sure that you do not use the option `-verbose` or `-silent` after `-easy-debug`. These options reduce or modify the information logged and might make debugging difficult.
2. Send this zipped file ending with `pscfg-output.zip` to MathWorks Technical Support for further debugging.

You can also create the zipped file with debug information during every `polyspace-configure` run by creating an environment variable `PS_CONFIGURE_OPTIONS` and setting its value to:

```
-easy-debug pathToFolder
```

where *pathToFolder* is the folder where you want to store debug information.

Verification Result

If you are having trouble understanding a result, see the results review guidelines in “Run-Time Checks”.

If you are still having trouble understanding the result, contact technical support with the following information:

- The analysis log.

  The analysis log is a text file generated in your results folder and titled `Polyspace_version_project_date_time.log`. It contains the options used for the analysis and other relevant information.
- The source files related to the result or the complete results folder if possible.

  If you cannot provide the source files:

  - Try to provide a screenshot of the relevant source code from the **Source** pane on the Polyspace user interface.
  - Try to reproduce the problem with a different code. Provide that code to technical support.

Provide Polyspace Analysis Statistics File (Optional)

Depending on your issue, you might be asked to provide the `.stats.zip` file. The file is located in your results folder and contains statistics about your analysis, such as options used, time taken by the different phases of the analysis, and the memory consumed by the different processes that ran during the analysis. The file contains no identifying information about your code.

See Also

Related Examples

- “Contact Technical Support About Polyspace Access Issues” on page 35-5

Fix Error: Polyspace Cannot Find Server

Message

```
Error: Cannot instantiate Polyspace cluster
| Check the -scheduler option validity or your default cluster profile
| Could not contact an MJS lookup service using the host computer_name.
    The hostname, computer_name, could not be resolved.
```

Possible Cause

Polyspace uses information provided in the preferences of a Polyspace desktop product to locate the server. If this information is incorrect, the software cannot locate the server.

Solution

Open the user interface of the Polyspace desktop product. Check if the server information provided is correct.

1. Select **Tools** > **Preferences**.
2. Select the **Server Configuration** tab. Check your server information.

   For instance, the entry in **Job scheduler host name** must match the host name of the computer that forms the head node of the MATLAB Parallel Server cluster. For more information, see “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

Fix Error: Job Manager Cannot Write to Database

Message

`Unable to write data to the job manager database`

Possible Cause

If the computer that forms the head node of the MATLAB Parallel Server cluster cannot send data to the client computer, you see this error. The most likely reasons for the remote computer being unable to connect to the client computer are:

- Firewalls do not allow traffic from the MATLAB Job Scheduler to the client.
- The MATLAB Job Scheduler cannot resolve the short hostname of the client computer.

Workaround

Add localhost IP to configuration.

1. In the user interface of the Polyspace desktop products, select **Tools** > **Preferences**.
2. On the **Server Configuration** tab, in the **Localhost IP address** field, enter the IP address of your local computer.

To retrieve your IP address:

- Windows
  1. Open **Control Panel** > **Network and Sharing Center**.
  2. Select your active network.
  3. In the Status window, click **Details**. Your IP address is listed under **IPv4 address**.
- Linux — Run the `ifconfig` command and find the `inet addr` corresponding to your network connection.
- Mac — Open **System Preferences** > **Network**.

See Also

Related Examples

- “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”
- “Connection Problems Between the Client and MATLAB Job Scheduler” (Parallel Computing Toolbox)

Resolve Error: No Compilation Unit Detected in Your Build

Issue

You can automatically create a Polyspace project or options file by running `polyspace-configure` on your build command at the command line or in the Polyspace user interface. `polyspace-configure` executes your build command, tracks the processes executed, extracts the compiler command invocations from the tracked processes, and determines source files to add to a Polyspace project. If your compiler command does not execute during the build or cannot be found, you might see this error:

```
No compilation unit detected in your build.
```

The error indicates that `polyspace-configure` could not find source files to add to a Polyspace project or options file.

Possible Solutions

Check for Incremental Build

Most build systems perform an incremental build by default. The build commands rebuild only the sources that changed since the last rebuild. For instance, if you build your sources by using `make` or its equivalent, the command rebuilds only those targets in the makefile that are out-of-date. If you run `polyspace-configure` on a build command and none of your sources have changed since the last rebuild, your compiler is not invoked and you see the preceding error. Even if some of the sources have changed, your compiler might rebuild only those sources, leaving your Polyspace project incomplete.

To avoid the error, when running `polyspace-configure`, perform a full or clean build of your sources. For instance, when building by using `make`, you can use the flag `-B` or `--always-make` to rebuild all targets in the makefile.

Check for Compiler Caching

Using a compiler cache is equivalent to performing an incremental build. Compiler caches speed up compilation by caching results of previous compilations. If the same compilation is repeated, the cached results are used instead of a fresh compilation. In this case, `polyspace-configure` cannot detect a compilation because the actual compiler commands are not invoked.

To work around the error, disable any compiler cache that you might be using just before running `polyspace-configure` on your build command. You can reenable the caching immediately after running `polyspace-configure`. For instance, if you use Ccache, you can disable the caching on the current shell by entering:

```
export CCACHE_DISABLE=1
```

Check for Antivirus Software

The `polyspace-configure` command works by tracking the processes executed by your build command. Certain antivirus software can block this tracking. `polyspace-configure` keeps a known list of antivirus software that block tracking and shows a warning if any of the software is detected. You might be using an antivirus software outside this known list and not see the warning but see a later error instead.

To avoid the issue, temporarily disable your antivirus software when running `polyspace-configure`. Some antivirus software allow you to specify a list of processes that must not be blocked. You might be able to work around the issue by specifying Polyspace processes in your allowlist. For details, see "Fix Errors or Slow Polyspace Runs from Disk Defragmentation and Anti-virus Software" on page 34-93.

Check for Unsupported Compilers

The `polyspace-configure` command supports the same compilers as Polyspace. Check if your compiler is supported in `Compiler (-compiler)`. For each supported compiler, `polyspace-configure` can recognize a known set of compiler invocation commands. If your compiler is not supported or is supported but uses an invocation command outside this known set, `polyspace-configure` fails to recognize the compiler invocation and produces the preceding error.

If your compiler is supported but uses a nonstandard invocation command, or is closely related to a supported compiler, you might be able to extend support to your compiler command. See "Create Polyspace Projects from Build Systems That Use Unsupported Compilers" on page 34-25. In all other cases, contact MathWorks Technical Support. See "Contact Technical Support About Issues with Running Polyspace" on page 34-18.

Check for Source Exclusions

You can use the option `-include-sources` or `-exclude-sources` with the `polyspace-configure` command to include or exclude certain sources from the generated Polyspace project or options file. If you use these options, make sure to check their arguments and ensure that you have not accidentally excluded more source files than you intend.

See also "Select Files for Polyspace Analysis Using Pattern Matching".

See Also
`polyspace-configure`

Related Examples

- "Requirements for Project Creation from Build Systems" on page 13-23
- "Create Polyspace Projects from Build Systems That Use Unsupported Compilers" on page 34-25

Create Polyspace Projects from Build Systems That Use Unsupported Compilers

Issue

Your compiler is not supported for automatic project creation from build commands.

Cause

For automatic project creation from your build system, your compiler configuration must be available to Polyspace. Polyspace provides a compiler configuration file only for certain compilers.

For information on which compilers are supported, see "Requirements for Project Creation from Build Systems" on page 13-23.

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 34-29.
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

`<compiler_names><name> ...`
 	
`</name><compiler_names>`
	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.
	
	You must not specify the linker binary inside the `<name>...</name>` elements.
	
	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 `-compiler-config` *`my_compiler`*`.xml` when tracing the build so that the software explicitly uses your compiler configuration file.
		• `gcc`
		• `gpp`

`<include_options><opt> ...`
 	
`</opt></include_options>`
	The option that you use with your compiler to specify include folders.
	
	To specify options where the argument immediately follows the option, use an `isPrefix` attribute for `opt` and set it to `true`.
		`-I`

`<system_include_options>`
 	
`<opt> ... </opt>`
 	
`</system_include_options>`
	The option that you use with your compiler to specify system headers.
	
	To specify options where the argument immediately follows the option, use an `isPrefix` attribute for `opt` and set it to `true`.
		`-isystem`

`<preinclude_options><opt> ...`
 	
`</opt></preinclude_options>`
	The option that you use with your compiler to force inclusion of a file in the compiled object.
	
	To specify options where the argument immediately follows the option, use an `isPrefix` attribute for `opt` and set it to `true`.
		`-include`

XML Element	Content Description	Content Example for GNU C Compiler

`<define_options><opt> ...`
 	
`</opt></define_options>`
	The option that you use with your compiler to predefine a macro.
	
	To specify options where the argument immediately follows the option, use an `isPrefix` attribute for `opt` and set it to `true`.
		`-D`

`<undefine_options><opt> ...`
 	
`</opt></undefine_options>`
	The option that you use with your compiler to undo any previous definition of a macro.
	
	To specify options where the argument immediately follows the option, use an `isPrefix` attribute for `opt` and set it to `true`.
		`-U`

`<semantic_options><opt> ...`
 	
`</opt></semantic_options>`
	The options that you use to modify the compiler behavior. These options specify the language settings to which the code must conform.
	
	You can use the `isPrefix` attribute to specify multiple options that have the same prefix and the `numArgs` attribute to specify options with multiple arguments. For instance:
	
	• Instead of
	
	`<opt>-m32</opt>`
	`<opt>-m64</opt>`
	
	You can write `<opt isPrefix="true">-m</opt>`.
	• Instead of
	
	`<opt>-std=c90</opt>`
	`<opt>-std=c99</opt>`
	
	You can write `<opt numArgs="1">-std</opt>`. If your makefile uses `-std c90` instead of `-std=c90`, this notation also supports that usage.
		• `-ansi`
		• `-std =C90`
		• `-std =c++11`
		• `-fun signed -char`

XML Element	Content Description	Content Example for GNU C Compiler
`<compiler> ... </compiler>`	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. For the complete list of compilers available, see `Compiler (-compiler)`.	`gnu4.7`

`<preprocess_options_list>`
 	
`<opt> ... </opt>`
 	
`</preprocess_options_list>`
	The options that specify how your compiler generates a preprocessed file.
	
	You can use the macro `$(OUTPUT_FILE)` if your compiler does not allow sending the preprocessed file to the standard output. Instead it defines the preprocessed file internally.
		`-E`
		
		For an example of the `$(OUTPUT_FILE)` macro, see the existing compiler configuration file `cl2000.xml`.

`<preprocessed_output_file> ... </preprocessed_output_file>`	The name of file where the preprocessed output is stored. You can use the following macros when the name of the preprocessed output file is adapted from the source file: • `$(SOURCE_FILE)`: Source file name • `$(SOURCE_FILE_EXT)`: Source file extension • `$(SOURCE_FILE_NO_EXT)`: Source file name without extension For instance, use `$(SOURCE_FILE_NO_EXT).pre` when the preprocessor file name has the same name as the source file, but with extension `.pre`.	For an example of this element, see the existing compiler configuration file `xc8.xml`.

`<src_extensions><ext> ...`
 	
`</ext></src_extensions>`
	The file extensions for source files.	• `c`
		• `cpp`
		• `c++`

XML Element	Content Description	Content Example for GNU C Compiler

`<obj_extensions><ext> ...`
 	
`</ext></obj_extensions>`
	The file extensions for object files.	

`<precompiled_header_extensions> ...`
 	
`</precompiled_header_extensions>`
	The file extensions for precompiled headers (if available).	

`<polyspace_extra_options_list>`
`   <opt> ... </opt>`
`   <opt> ... </opt>`
`</polyspace_extra_options_list>`
	Additional options that are used for the subsequent analysis.
	
	For instance, to avoid compilation errors in the subsequent analysis due to non-ANSI extension keywords, enter `-D` *`keyword`*`=`*`value`*, for example:
	
	`<polyspace_extra_options_list>`
	`    <opt>-D MACRO1</opt>`
	`    <opt>-D MACRO2=VALUE</opt>`
	`</polyspace_extra_options_list>`
	
	For more information, see `Preprocessor definitions (-D)`.	

Mapping Between Existing Configuration Files and Compiler Names

Select the configuration file in *`polyspaceroot`*`\polyspace\configure\compiler_configuration\` that most closely resembles the configuration of your compiler. Use the following table to map compilers to their configuration files.

Compiler Name	Vendor	XML File
ARM	ARM Keil	`armcc.xml`
		`armclang.xml`
Visual C++	Microsoft	`cl.xml`
Clang	Not applicable	`clang.xml`
CodeWarrior	NXP	`cw_ppc.xml`
		`cw_s12z.xml`
`cx6808`	Cosmic	`cx6808.xml`
		`cosmic.xml`
Diab	Wind River	`diab.xml`
gcc	Not applicable	`gcc.xml`
Green Hills	Green Hills Software	`ghs_arm.xml`
		`ghs_arm64.xml`
		`ghs_i386.xml`

Compiler Name	Vendor	XML File
		ghs_ppc.xml
		ghs_rh850.xml
		ghs_tricore.xml
IAR Embedded Workbench	IAR	iar.xml
		iar-arm.xml
		iar-avr.xml
		iar-msp430.xml
		iar-rh850.xml
		iar-riscv.xml
		iar-rl78.xml
Renesas	Renesas	renesas-rh850.xml
		renesas-rl78.xml
		renesas-rx.xml
		renesas-sh.xml
TASKING®	Altium	tasking.xml
		tasking-166.xml
		tasking-850.xml
		tasking-arm.xml
Tiny C	Not applicable	tcc.xml
TM320 and its derivatives	Texas Instruments	ti_arm.xml
		ti_c28x.xml
		ti_c6000.xml
		ti_msp430.xml
xc8 (PIC)	Microchip	microchip.xml

Fix Slow Build Process When Polyspace Traces Build

Issue

In some cases, your build process can run slower when Polyspace traces the build.

Cause

Polyspace caches information in files stored in the system temporary folder, such as `C:\Users\User_Name\AppData\Local\Temp`, in Windows. Your build can take a long time to perform read/write operations to this folder. Therefore, the overall build process is slow.

Solution

You can work around the slow build process by changing the location where Polyspace stores cache information. For instance, you can use a cache path local to the drive from which you run build tracing. To create and use a local folder `ps_cache` for storing cache information, use the advanced option `-cache-path ./ps_cache`.

- If you trace your build from the Polyspace user interface, enter this flag in the field **Add advanced configure options**.
- If you trace your build from the DOS/ UNIX or MATLAB command line, use this flag with the `polyspace-configure` command.

For more information, see `polyspace-configure`.

Check if Polyspace Supports Build Scripts

Issue

This topic is relevant only if you are creating a Polyspace project in Windows from your build scripts.

When Polyspace traces your build script in a Windows console application other than `cmd.exe`, the command fails. However, the build command by itself executes to completion.

For instance, your build script executes to completion from the Cygwin shell. However, when Polyspace traces the build, the build script throws an error.

Possible Cause

When you launch a Windows console application, your environment variables are appropriately set. Alternate console applications such as the Cygwin shell can set your environment differently from `cmd.exe`.

Polyspace attempts to trace your build script with the assumption that the script runs to completion in `cmd.exe`. Therefore, even if your script runs to completion in the alternate console application, when Polyspace traces the build, the script can fail.

Solution

Make sure that your build script executes to completion in the `cmd.exe` interface. If the build executes successfully, create a wrapper `.bat` file around your script and trace this file.

For instance, before you trace a build command that executes to completion in the Cygwin shell, launch the Cygwin shell from `cmd.exe` and then run your build script. For example, if you use a script `build.sh` to build your code, enter the following command at the DOS command line:

```
cmd.exe /C "C:\cygwin64\bin\bash.exe" --login -c build.sh
```

If the steps do not execute to completion, Polyspace cannot trace your build.

If the steps complete successfully, trace the build command after launching it from `cmd.exe`. For instance, on the command-line, do the following to create a Polyspace options file.

1. Enter your build commands in a `.bat` file.

   ```
   rem @echo off
   cmd.exe /C "C:\cygwin64\bin\bash.exe" --login -c build.sh
   ```

   Name the file, for instance, `launching.bat`.
2. Trace the build commands in the `.bat` file and create a Polyspace options file.

   ```
   "C:\Program Files\MATLAB\R2023a\polyspace\bin\polyspace-configure.exe"
             -output-options-file myOptions.txt launching.bat
   ```

You can now run `polyspace-code-prover` or `polyspace-code-prover-server` on the options file.

Troubleshoot Project Creation from MinGW Build

Issue

You create a project from a MinGW build, but get an error when running an analysis on the project. The error message comes from using one of these keywords: `__declspec`, `__cdecl`, `__fastcall`, `__thiscall` or `__stdcall`.

Cause

When you create a project from a MinGW build, the project uses a GNU compiler. Polyspace does not recognize these keywords for the GNU compilers.

Solution

Replace these keywords with equivalent keywords just for the purposes of analysis.

Before analysis, for the option `Preprocessor definitions (-D)`, enter:

- `__declspec(x)=__attribute__((x))`
- `__cdecl=__attribute__((__cdecl__))`
- `__fastcall=__attribute__((__fastcall__))`
- `__thiscall=__attribute__((__thiscall__))`
- `__stdcall=__attribute__((__stdcall__))`

If you are running Polyspace on the command line in a UNIX shell, add double quotes around the `-D` option. For instance, use:

```
"-D __cdecl=__attribute__((__cdecl__))"
```

Troubleshoot Project Creation from Visual Studio Build

You can run the `polyspace-configure` command to create a Polyspace project or options file from a Visual Studio build. The command monitors the processes executed during the build and extracts information required for the project or options file

You can trace your Visual Studio build in one of the following ways:

- Build your Visual Studio project completely at the command line with `msbuild` while tracing this build with `polyspace-configure`.

  In this workflow, you run `polyspace-configure` on an `msbuild` command with a Visual Studio project (`.vcxproj`) file. For instance, in a Visual Studio 2019 developer prompt, enter the following:

  ```
  polyspace-configure msbuild TestProject.vcxproj /t:Rebuild
  ```
- Build your Visual Studio project in the Visual Studio IDE while tracing this build with `polyspace-configure`.

  Run `polyspace-configure` on the `devenv.exe` executable to open the Visual Studio IDE, build your project or solution within the IDE, and then close the IDE.

  See “Create Polyspace Projects from Visual Studio Build” on page 2-9.

If running `polyspace-configure` on the `msbuild` command does not work properly, do the following:

1. Stop the `msbuild` process.
2. Set the environment variable `MSBUILDDISABLENODEREUSE` to 1.
3. Restart `polyspace-configure` on `msbuild`, this time using the `/nodereuse:false` option. For instance:

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

See Also
`polyspace-configure`

Resolve polyspace-autosar Error: Could Not Find Include File

Issue

When creating a Polyspace project from an AUTOSAR description, by default, the software searches only in the source folder for `#include`-d files. If an include file is not present directly in the source folder, Polyspace cannot find it. For instance, the missing include file can be in a subfolder of the source folder.

You see a warning like this when creating a Polyspace project from AUTOSAR XML and source files:

```
Could not find include file "MemMap.h"
```

If you use variables or functions declared in the missing include file, you can also see errors later.

Possible Solutions

If you want to expand the search path for include files, explicitly add new folders.

- In the Polyspace user interface, use the field **Specify additional include folders**.

  See “Run Polyspace on AUTOSAR Code” on page 8-14.
- At the command-line, use the option `-I`.

  See `polyspace-autosar`.

You can find the possible include folders to add in several ways:

- If an include file is in a subfolder of the source code folder, the analysis proposes a resolution hint with one or more include folders that might contain the missing include file. To see the resolution hints, in the file `psar_project.xhtml`, click the button on the upper left, then click **Behaviors**. On the **Behaviors** tab, below the errors in the code extraction phase, click the link to see a summary of code-extraction diagnostics with possible resolution hints.

Extract implementation code for **89** AUTOSAR behaviors with proof artifacts:

- noRunnableImplementation (30)
- error_noRunnableImplementationTopFileError (3)
- error_atLeastOneRunnableInFileThatDoesNotCompile (23)
- subsetOfRunnablesImplementation (3)
- allRunnablesImplementation (30)

See summary of code-extraction diagnostics with possible resolution hints

Execution reported errors and warnings. Reported errors See detailed log messages

  You can see resolution hints, that is, possible include folders to add, that would resolve some of the missing include files.

Instead of fixing individual code extraction errors using the resolution hints, you can also download a file with all options that implement the hints. On the summary page, click the link **Download polyspace-autosar options**.

Summary of polyspace-autosar code-extraction diagnostics

Lists diagnostics that are reported when extracting the implementation-code of one or more AUTOSAR behaviors.
Each diagnostic may have "resolution-hints" which are specific to the class of error.
Resolution-hints can translate to polyspace-autosar options that you may add to your project Download polyspace-autosar options

You can use the downloaded text file with the `polyspace-autosar` option `-options-file` to implement the resolution hints in one shot.

- If you use a build command for compilation, you can extract compilation options such as path to includes from your build command. See “Run Polyspace on AUTOSAR Code Using Build Command” on page 8-31.

You might also simply know the architecture of the system to locate the missing include folders.

See Also

`polyspace-autosar`

Related Examples

- “Run Polyspace on AUTOSAR Code” on page 8-14
- “Run Polyspace on AUTOSAR Code Using Build Command” on page 8-31
- “Troubleshoot Polyspace Analysis of AUTOSAR Code” on page 8-22

Resolve polyspace-autosar Error: Conflicting Universal Unique Identifiers (UUIDs)

Issue

If multiple elements in an AUTOSAR description contain the same Universal Unique Identifier (UUID) or a single element contains multiple UUIDs, one of these errors can occur when creating a Polyspace project from the AUTOSAR XML files:

- `Elements "/pkg/swc002/bhv/twosec" and "/pkg/swc002/bhv/step" in file $file{C:/AUTOSAR/arxml/mSwc002_component.arxml}{332} have the same UUID "5bdd54d5-50ae-4ad3-bdea-e0b0ab2bcab6". Each of these elements should have its own unique UUID.`
- `'Element "/AUTOSAR" has both UUID "ECUS:6b411924-70da-40a5-85f5-65d5630ea0cb" and "ECUS:48ea040a-c40d-4ee0-ae61-8a6ccc9cb18d". You should specify only one UUID.`

Possible Solutions

Investigate why multiple elements have the same UUID, or the same element has two different UUID-s. Fix the issue if possible.

If you do not own the AUTOSAR XML with the conflicting UUID-s or do not want to fix the issue because it represents work in progress, use the options `-Eno-autosar-xmlReaderSameUuidForDifferentElements` and `-Eno-autosar-xmlReaderTooManyUuids`. The analysis ignores the issue of conflicting UUID-s and continues with a warning. For conflicting UUID-s, the analysis stores the last element read.

The subsequent analyses continue to use the warning mode. To revert back to the error mode, use the option `-Eautosar-xmlReaderSameUuidForDifferentElements` and `-Eautosar-xmlReaderTooManyUuids`.

See Also

`polyspace-autosar`

Related Examples

- "Run Polyspace on AUTOSAR Code" on page 8-14
- "Troubleshoot Polyspace Analysis of AUTOSAR Code" on page 8-22

Resolve polyspace-autosar Error: Data Type Not Recognized

Issue

When creating a Polyspace project from an AUTOSAR description, the software parses your AUTOSAR XML specifications and imports the data types that are required by the Software Components in the scope of verification. If your code uses a data type that is not in the Software Component specification, the analysis does not recognize this data type.

You see an error such as:

```
Identifier "LaneDetectionVar" is undefined
```

when creating a Polyspace project from AUTOSAR XML and source files. The error suggests that a data type used in your source code is not recognized.

Possible Solutions

You can force import of data types that are not defined for Software Components that you are verifying. Use the option `-autosar-datatype`. See `polyspace-autosar`.

You can find the already imported data types using the file `autosar_model_key_elements.html` in the `AUTOSAR` subfolder of your project folder. In the `DataTypes` section of the HTML, the file shows:

- Automatically imported data types using this format:

indirect	`pkg.types.app.Array_2_n320to320`
indirect	`pkg.types.app.Boolean`

  The text `indirect` in the first column indicates that the data types are automatically imported.
- Explicitly imported data types using this format:

name	`tst003.typ.app.Boolean`

  The text `name` in the first column indicates that the data type `tst003.typ.app.Boolean` is explicitly imported for the analysis.

In some cases, the analysis proposes a resolution hint using additional data types imported from the ARXML as a possible match for the unrecognized data type. To see the resolution hints, in the file `psar_project.xhtml`, click the ☰ button on the upper left, then click **Behaviors**. On the **Behaviors** tab, below the errors in the code extraction phase, click the link to see a summary of code-extraction diagnostics with possible resolution hints.

Extract implementation code for **89** AUTOSAR behaviors with proof artifacts:

- noRunnableImplementation (30)
- error_noRunnableImplementationTopFileError (3)
- error_atLeastOneRunnableInFileThatDoesNotCompile (23)
- subsetOfRunnablesImplementation (3)
- allRunnablesImplementation (30)

See summary of code-extraction diagnostics with possible resolution hints

Execution reported errors and warnings. Reported errors See detailed log messages

You can see resolution hints, that is, possible data types to add, that would resolve some of the issues related to unrecognized data types.

Instead of fixing individual code extraction errors using the resolution hints, you can also download a file with all options that implement the hints. On the summary page, click the link **Download polyspace-autosar options**.

Summary of polyspace-autosar code-extraction diagnostics

Lists diagnostics that are reported when extracting the implementation-code of one or more AUTOSAR behaviors.
Each diagnostic may have "resolution-hints" which are specific to the class of error.
Resolution-hints can translate to polyspace-autosar options that you may add to your project Download polyspace-autosar options

You can use the downloaded text file with the `polyspace-autosar` option `-options-file` to implement the resolution hints in one shot.

See Also

`polyspace-autosar`

Related Examples

- "Run Polyspace on AUTOSAR Code" on page 8-14
- "Troubleshoot Polyspace Analysis of AUTOSAR Code" on page 8-22

Fix Polyspace Compilation Errors About Undefined Identifiers

Issue

Polyspace verification fails during the compilation phase with a message about undefined identifiers.

The message indicates that Polyspace cannot find a variable definition. Therefore, it cannot identify the variable type.

Possible Cause: Missing Files

The source code you provided does not contain the variable definition. For instance, the variable is defined in an include file that Polyspace cannot find.

If you `#include`-d the include file in your source code but did not add it to your Polyspace project, you see a previous warning:

```
Warning: could not find include file "my_include.h"
```

Solution

If the variable definition occurs in an include file, add the folder that contains the include file.

- In the user interface of the Polyspace desktop products, add the folder to your project.

  For more information, see "Add Source Files for Analysis in Polyspace Desktop User Interface" on page 2-2.
- At the command line, use the flag `-I` with the `polyspace-code-prover` command.

  For more information, see `-I`.

Possible Cause: Unrecognized Keyword

The variable represents a keyword that your compiler recognizes but is not part of the ANSI C standard. Therefore, Polyspace does not recognize it.

For instance, some compilers interpret `__SP` as a reference to the stack pointer.

Solution

If the variable represents a keyword that Polyspace does not recognize, replace or remove the keyword from your source code or preprocessed code.

If you remove or replace the keyword from the preprocessed code, you can avoid the compilation error while keeping your source code intact. You can do one of the following:

- Replace or remove each individual unknown keyword using an analysis option. Replace the compiler-specific keyword with an equivalent keyword from the ANSI C Standard.

  For information on the analysis option, see `Preprocessor definitions (-D)`.
- Declare the unknown keywords in a separate header file using `#define` directives. Specify that header file using an analysis option.

For information on the analysis option, see `Include (-include)`. For a sample header file, see “Gather Compilation Options Efficiently” on page 13-31.

Possible Cause: Declaration Embedded in #ifdef Statements

The variable is declared in a branch of an `#ifdef` *`macro_name`* preprocessor directive. For instance, the declaration of a variable `max_power` occurs as follows:

```
#ifdef _WIN32
  #define max_power 31
#endif
```

Your compilation toolchain might consider the macro *`macro_name`* as implicitly defined and execute the `#ifdef` branch. However, the Polyspace compilation might not consider the macro as defined. Therefore, the `#ifdef` branch is not executed and the variable `max_power` is not declared.

Solution

To work around the compilation error, do one of the following:

- Use **Target & Compiler** options to directly specify your compiler. For instance, to emulate a Visual C++ compiler, set the **Compiler** to `visual12.0`. See “Target and Compiler”.
- Define the macro explicitly using the option `Preprocessor definitions (-D)`.

Note If you create a Polyspace by tracing your build commands, most **Target & Compiler** options are automatically set.

Possible Cause: Project Created from Non-Debug Build

This can be a possible cause only if the undefined identifier occurs in an `assert` statement (or an equivalent Visual C++ macro such as `ASSERT` or `VERIFY`).

Typically, you come across this error in the following way. You create a Polyspace project from a build system in non-debug mode. When you run an analysis on the project, you face a compilation error from an undefined identifier in an `assert` statement. You find that the identifier `my_identifier` is defined in a `#ifndef NDEBUG` statement, for instance as follows:

```
#ifndef NDEBUG
int my_identifier;
#endif
```

The C standard states that when the `NDEBUG` macro is defined, all assert statements must be disabled.

Most IDEs define the `NDEBUG` macro in their build systems. When you build your source code in your IDE in non-debug mode, code in a `#ifndef NDEBUG` statement is removed during preprocessing. For instance, in the preceding example, `my_identifier` is not defined. If `my_identifier` occurs only in assert statements, it is not used either, because `NDEBUG` disables assert statements. You do not have compilation errors from undefined identifiers and your build system executes successfully.

Polyspace does not disable `assert` statements even if `NDEBUG` macro is defined because the software uses `assert` statements internally to enhance verification.

When you create a Polyspace project from your build system, if your build system defines the `NDEBUG` macro, it is also defined for your Polyspace project. Polyspace removes code in a `#ifndef NDEBUG` statement during preprocessing, but does not disable `assert` statements. If `assert` statements in your code rely on the code in a `#ifndef NDEBUG` statement, compilation errors can occur.

In the preceding example:

- The definition of `my_identifier` is removed during preprocessing.
- `assert` statements are not disabled. When `my_identifier` is used in an `assert` statement, you get an error because of undefined identifier `my_identifier`.

Solution

To work around this issue, create a Polyspace project from your build system in debug mode. When you execute your build system in debug mode, `NDEBUG` is not defined. When you create a Polyspace project from this build, `NDEBUG` is not defined for your Polyspace project.

Depending on your project settings, use the option that enables building in debug mode. For instance, if your build system is `gcc`-based, you can define the `DEBUG` macro and undefine `NDEBUG`:

```
gcc -DDEBUG=1 -UNDEBUG *.c
```

Alternatively, you can disable the `assert` statements in your preprocessed code using the option `Preprocessor definitions (-D)`. However, Polyspace will not be able to emulate the `assert` statements.

Fix Polyspace Compilation Errors About Unknown Function Prototype

Issue

During the compilation phase, the software displays a warning or error message about unknown function prototype.

```
the prototype for function 'myfunc' is unknown
```

The message indicates that Polyspace cannot find a function prototype. Therefore, it cannot identify the data types of the function argument and return value, and has to infer them from the calls to the function.

To determine the data types for such functions, Polyspace follows the C99 Standard (ISO/IEC 9899:1999, Chapter 6.5.2.2: Function calls).

- The return type is assumed to be `int`.
- The number and type of arguments are determined by the first call to the function. For instance, if the function takes one `double` argument in the first call, for subsequent calls, the software assumes that it takes one `double` argument. If you pass an `int` argument in a subsequent call, a conversion from `int` to `double` takes place.

During the linking phase, if a mismatch occurs between the number or type of arguments or the return type in different compilation units, the verification stops. For more information, see “Fix Polyspace Linking Errors About Conflicting Declarations in Different Translation Units” on page 34-60.

Cause

The source code you provided does not contain the function prototype. For instance, the function is declared in an include file that Polyspace cannot find.

If you `#include`-d the include file in your source code but did not add it to your Polyspace project, you see a previous warning:

```
Warning: could not find include file "my_include.h"
```

Solution

Search for the function declaration in your source repository.

If you find the function declaration in an include file, add the folder that contains the include file.

- In the user interface of the Polyspace desktop products, add the folder to your project.

  For more information, see “Add Source Files for Analysis in Polyspace Desktop User Interface” on page 2-2.
- At the command line, use the flag `-I` with the `polyspace-code-prover` command.

  For more information, see `-I`.

Fix Polyspace Compilation Errors Related to #error Directive

Issue

The analysis stops with a message containing a `#error` directive. For instance, the following message appears: `#error directive: !Unsupported platform; stopping!`.

Cause

You typically use the `#error` directive in your code to trigger a fatal error in case certain macros are not defined. Your compiler implicitly defines the macros, therefore the error is not triggered when you compile your code. However, the default Polyspace compilation does not consider the macros as defined, therefore, the error occurs.

For instance, in the following example, the `#error` directive is reached only if the macros `__BORLANDC__`, `__VISUALC32__` or `__GNUC__` are not defined. If you use a GNU C compiler, for instance, the compiler considers the macro `__GNUC__` as defined and the error does not occur. However, if you use the default Polyspace compilation, it does not consider the macros as defined.

```
#if defined(__BORLANDC__) || defined(__VISUALC32__)
#define MYINT int
#elif defined(__GNUC__)
#define MYINT long
#else
#error !Unsupported platform; stopping!
#endif
```

Solution

For successful compilation, do one of the following:

- Specify a compiler such as `visual12.0` or `gnu4.9`. Specifying a compiler defines some of the compilation flags for the analysis.

  For more information, see `Compiler (-compiler)`.
- If the available compiler options do not match your compiler, explicitly define one of the compilation flags `__BORLANDC__`, `__VISUALC32__`, or `__GNUC__`.

  For more information, see `Preprocessor definitions (-D)`.

Fix Polyspace Compilation Errors About Large Objects

Issue

The analysis stops during compilation with a message indicating that an object is too large.

Cause

The error happens when the software detects an object such as an array, union, structure, or class, that is too big for the pointer size of the selected target.

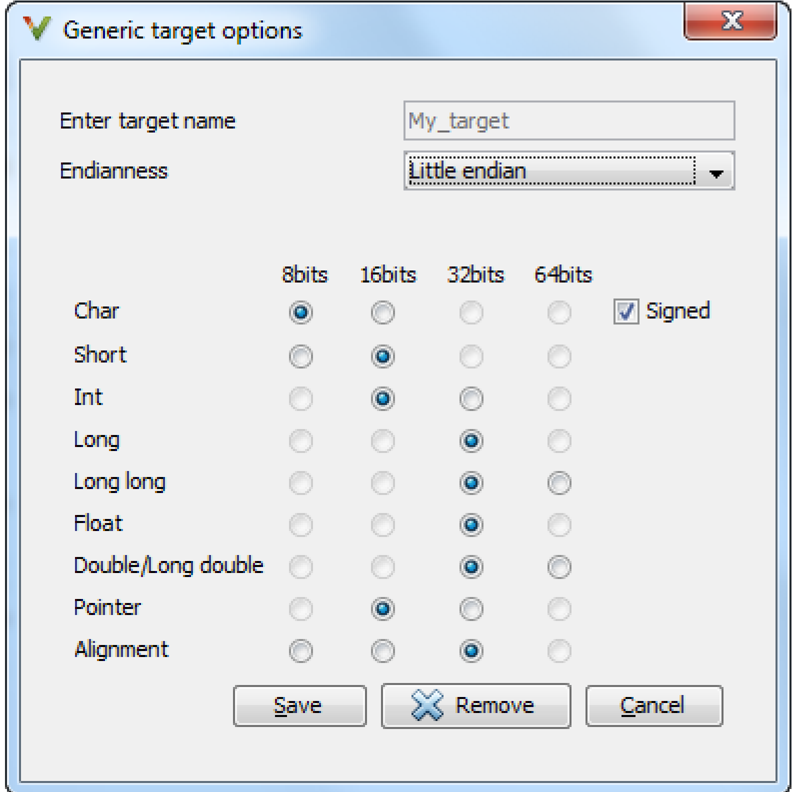
For instance, you get the message, `Limitation: struct or union is too large` in the following example. You specify a pointer size of 16 bits. The maximum object size allocated to a pointer, and therefore the maximum allowed size for an object, can be 2^{16}-1 bytes. However, you declare a structure as follows:

- ```
  struct S
  {
    char tab[65536];
  }s;
  ```
- ```
  struct S
  {
    char tab[65534];
    int val;
  }s;
  ```

Solution

1 Check the pointer size that you specified through your target processor type. For more information, see `Target processor type (-target)`.

   For instance, in the following, the pointer size for a custom target `My_target` is 16 bits.



2 Change your code or specify a different pointer size.

For instance, you can:

- Declare an array of smaller size in the structure.

  If you are using a predefined target processor type, the pointer size is likely to be the same as the pointer size on your target architecture. Therefore, your declaration might cause errors on your target architecture.
- Change the pointer size of the target processor type that you specified, if possible.

  Otherwise, specify another target processor type with larger pointer size or define your own target processor type. For more information on defining your own processor type, see `Generic target options`.

Note Polyspace imposes an internal limit of 128 MB on the size of data structures. Even if your target processor type specification allows data structures of larger size, this internal limit constrains the data structure sizes.

Fix Polyspace Compilation Errors Related to Generic Compiler

If you use a generic compiler, you can encounter this issue. For more information, see `Compiler (-compiler)`.

Issue

The analysis stops with an error message related to a non-ANSI C keyword, for instance, `data` or attributes such as `__attribute__((weak))`.

Depending on the location of the keyword, the error message can vary. For instance, this line causes the error message: `expected a ";"`.

```
data int tab[10];
```

Cause

The generic Polyspace compiler supports only ANSI C keywords. If you use a language extension, the generic compiler does not recognize it and treats the keyword as a regular identifier.

Solution

Specify your compiler by using the option `Compiler (-compiler)`.

If your compiler is not directly supported or is not based on a supported compiler, you can use the generic compiler. To work around the compilation errors:

- If the keyword is related to memory modelling, remove it from the preprocessed code. For instance, to remove the `data` keyword, enter `data=` for the option `Preprocessor definitions (-D)`.
- If the keyword is related to an attribute, remove attributes from the preprocessed code. Enter `__attribute__(x)=` for the option `Preprocessor definitions (-D)`.

  If your code has this line:

  ```
  void __attribute__ ((weak)) func(void);
  ```

  And you remove attributes, the analysis reads the line as:

  ```
  void func(void);
  ```

When you use these workarounds, your source code is not altered.

Fix Polyspace Compilation Errors Related to GNU Compiler

If you choose `gnu` for the option `Compiler (-compiler)`, you can encounter this issue.

Issue

The Polyspace analysis stops with a compilation error.

Cause

You are using certain advanced compiler-specific extensions that Polyspace does not support. See “Limitations”.

Solution

For easier portability of your code, avoid using the extensions.

If you want to use the extensions and still analyze your code, wrap the unsupported extensions in a preprocessor directive. For instance:

```
#ifdef POLYSPACE
    // Supported syntax
#else
    // Unsupported syntax
#endif
```

For regular compilation, do not define the macro `POLYSPACE`. For Polyspace analysis, enter `POLYSPACE` for the option `Preprocessor definitions (-D)`.

If the compilation error is related to assembly language code, use the option `-asm-begin -asm-end`.

Fix Polyspace Compilation Errors Related to Visual Compilers

The following messages appear if the compiler is based on a Visual compiler. For more information, see `Compiler (-compiler)`.

Import Folder

When a Visual application uses `#import` directives, the Visual C++ compiler generates a header file with extension `.tlh` that contains some definitions. To avoid compilation errors during Polyspace analysis, you must specify the folder containing those files.

Original code:

```
#include "stdafx.h"
#include <comdef.h>
#import <MsXml.tlb>
MSXML::_xml_error e ;
MSXML::DOMDocument* doc ;
int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

Error message:

```
"../sources/ImportDir.cpp", line 7: catastrophic error: could not
open source file "./MsXml.tlh"
    #import <MsXml.tlb>
```

The Visual C++ compiler generates these files in its “build-in” folder (usually Debug or Release). In order to provide those files:

- Build your Visual C++ application.
- Specify your build folder for the Polyspace analysis.

pragma Pack

Using a different value with the compile flag (`#pragma pack`) can lead to a linking error message.

Original code:

test1.cpp	type.h	test2.cpp

`#pragma pack(4)`
 	
`#include "type.h"`
	`struct A`
	`{`
	`    char c ;`
	`    int i ;`
	`} ;`
		`#pragma pack(2)`
		
		`#include "type.h"`

Error message:

```
Pre-linking C++ sources ...
"../sources/type.h", line 2: error: declaration of class "A" had
a different meaning during compilation of "test1.cpp"
(class types do not match)
    struct A
```

```
          ^
          detected during compilation of secondary translation unit
"test2.cpp"
```

To continue the analysis, use the option `Ignore pragma pack directives (-ignore-pragma-pack)`.

C++/CLI

Polyspace does not support Microsoft C++/CLI, a set of language extensions for .NET programming.

You can get errors such as:

```
error: name must be a namespace name
|          using namespace System;
```

Or:

```
error: expected a declaration
|          public ref class Form1 : public System::Windows::Forms::Form
```

Fix Polyspace Compilation Errors Related to Keil or IAR Compiler

If you use the compiler, Keil or IAR, you can encounter this issue. For more information, see `Compiler (-compiler)`.

Missing Identifiers

Issue

The analysis stops with the error message, `expected an identifier`, as if an identifier is missing. However, in your source code, you can see the identifier.

Cause

If you select Keil or IAR as your compiler, the software removes certain keywords during preprocessing. If you use these keywords as identifiers such as variable names, a compilation error occurs.

For a list of keywords that are removed, see "Supported Keil or IAR Language Extensions" on page 13-26.

Solution

Specify that Polyspace must not remove the keywords during preprocessing. Define the macros `__PST_KEIL_NO_KEYWORDS__` or `__PST_IAR_NO_KEYWORDS__`.

For more information, see `Preprocessor definitions (-D)`.

Fix Polyspace Compilation Errors Related to Diab Compiler

If you choose `diab` for the option `Compiler (-compiler)`, you can encounter this issue.

Issue

During Polyspace analysis, you see an error related to a keyword specific to the Diab compiler. For instance, you see an error related to the `restrict` keyword.

Cause

You typically use a compiler flag to enable the keyword. The Polyspace analysis does not enable these keywords by default. You have to make Polyspace aware of your compiler flags.

The Polyspace analysis does not enable these keywords by default to prevent compilation errors. Another user might not enable the keyword and instead use the keyword name as a regular identifier. If Polyspace treats the identifier as a keyword, a compilation error will occur.

Solution

Use the command-line option `-compiler-parameter` in your Polyspace analysis as follows. You use this command-line option to make Polyspace aware of your compiler flags. In the user interface of the Polyspace desktop products, you can enter the command-line option in the field `Other`. You can enter the option multiple times.

The argument of `-compiler-parameter` depends on the keyword that causes the error. Once you enable the keyword, do not use the keyword name as a regular identifier. For instance, once you enable the keyword `pixel`, do not use `pixel` as a variable name. The statement `int pixel = 1` causes a compilation error.

- `restrict` keyword:

  You typically use the compiler flag `-Xlibc-new` or `-Xc-new`. For your Polyspace analysis, use

  ```
  -compiler-parameter -Xc-new
  ```

  The following code will not compile with Polyspace unless you specify the compiler flag.

  ```
  int sscanf(const char *restrict, const char *restrict, ...);
  ```
- PowerPC AltiVec vector extensions such as the `vector` type qualifier:

  You typically use the compiler flag `-tPPCALLAV:`. For your Polyspace analysis, use

  ```
  -compiler-parameter -tPPCALLAV:
  ```

  The following code will not compile with Polyspace unless you specify the compiler flag.

  ```
  vector unsigned char vbyte;
  vector bool vbool;
  vector pixel vpx;

  int main(int argc, char** argv)
  {
  ```

```
  return 0;
}
```

- Extended keywords such as `pascal`, `inline`, `packed`, `interrupt`, `extended`, `__X`, `__Y`, `vector`, `pixel`, `bool` and others:

  You typically use the compiler flag `-Xkeywords=`. For your Polyspace analysis, use

  ```
  -compiler-parameter -Xkeywords=0xFFFFFFFF
  ```

  The following code will not compile with Polyspace unless you specify the above option:

  ```
  packed(4) struct s2_t {
      char b;
      int i;
  } s2;

  packed(4,2) struct s3_t {
      char b;
  } s3;

  int pascal foo = 4;

  int main(int argc, char** argv) {
      foo++;
      return 0;
  }
  ```

  Note that the Polyspace option only allows the code to be compiled. The analysis does not fully support the semantics behind the `packed` keyword.

Fix Polyspace Compilation Errors Related to Green Hills Compiler

If you choose `greenhills` for the option `Compiler (-compiler)`, you encounter this issue.

Issue

During Polyspace analysis, you see an error related to vector data types specific to Green Hills target `rh850`. For instance, you see an error related to identifier `__ev64_u16__`.

Cause

When compiling code using the Green Hills compiler with target `rh850`, to enable single instruction multiple data (SIMD) vector instructions, you specify two flags:

- `-rh850_simd`: You enable intrinsic functions that support SIMD vector instructions. The functions are defined in your compiler header files. These data types are available:
  - `__ev64_u16__`
  - `__ev64_s16__`
  - `__ev64_u32__`
  - `__ev64_s32__`
  - `__ev64_u64__`
  - `__ev64_s64__`
  - `__ev64_opaque__`
  - `__ev128_opaque__`
- `-rh850_fpsimd`: You enable intrinsic functions that support floating-point SIMD vector instructions. The functions are defined in your compiler header files. These data types are available:
  - `__ev128_f32__`
  - `__ev256_f32__`

The Polyspace analysis does not enable SIMD support by default. You must identify your compiler flags to Polyspace.

Solution

In your Polyspace analysis, use the command-line option `-compiler-parameter`. In the user interface, you can enter the command-line option in the `Other` field, under the **Advanced Settings** in the **Configuration** pane.

- `-rh850_simd`: For your Polyspace analysis, use

  ```
  -compiler-parameter -rh850_simd
  ```
- `-rh850_fpsimd`: For your Polyspace analysis, use

  ```
  -compiler-parameter -rh850_fpsimd
  ```

Note

- `__ev128_opaque__` is 16 bytes aligned in Polyspace.
- `__ev256_f32__` is 32 bytes aligned in Polyspace.

Fix Polyspace Compilation Errors Related to TASKING Compiler

If you choose `tasking` for the option `Compiler (-compiler)`, you can encounter this issue.

Issue

During Polyspace analysis, you see an error related to a Special Function Register data type.

Cause

When compiling with the TASKING compiler, you typically use the following compiler flags to specify where Special Function Register (SFR) data types are declared:

- `--cpu=`*`xxx`*: The compiler implicitly `#include`s the file `sfr/reg`*`xxx`*`.sfr` in your source files. Once `#include`-ed, you can use Special Function Registers (SFR-s) declared in that `.sfr` file.
- `--alternative-sfr-file`: The compiler uses an alternative SFR file instead of the regular SFR file. You can use Special Function Registers (SFR-s) declared in that alternative SFR file.

If you specify the TASKING compiler for your Polyspace analysis, the analysis makes the following assumptions about these compiler flags:

- `--cpu=`*`xxx`*: The analysis chooses a specific value of *`xxx`*. If you use a different value with your TASKING compiler, you can encounter an error during Polyspace analysis.

  The *`xxx`* value that the Polyspace analysis uses depends on your choice of `Target processor type (-target)`:

  - `tricore`: `tc1793b`
  - `c166`: `xc167ci`
  - `rh850`: `r7f701603`
  - `arm`: `ARMv7M`
- `--alternative-sfr-file`: The analysis assumes that you do not use an alternative SFR file. If you use one, you can encounter an error.

Solution

Use the command-line option `-compiler-parameter` in your Polyspace analysis as follows. You use this command-line option to make Polyspace aware of your compiler flags. In the user interface, you can enter the command-line option in the field `Other`. You can enter the option multiple times.

- `--cpu=`*`xxx`*: For your Polyspace analysis, use

  ```
  -compiler-parameter --cpu=xxx
  ```

  Here, *`xxx`* is the value that you use when compiling with your compiler.
- `--alternative-sfr-file`: For your Polyspace analysis, use

  ```
  -compiler-parameter --alternative-sfr-file
  ```

  If you still encounter an error because Polyspace is not able to locate your `.asfr` file, explicitly `#include` your `.asfr` file in the preprocessed code using the option `Include (-include)`.

Typically, the path to the file is `Tasking_C166_INSTALL_DIR\include\sfr\regCPUNAME.asfr`. For instance, if your TASKING compiler is installed in `C:\Program Files\Tasking\C166-VX_v4.0r1\` and you use the CPU-related flag `-Cxc2287m_104f` or `--cpu=xc2287m_104f`, the path is `C:\Program Files\Tasking\C166-VX_v4.0r1\include\sfr\regxc2287m.asfr`.

You can also encounter the same issue with alternative sfr files when you trace your build command. For more information, see “Requirements for Project Creation from Build Systems” on page 13-23.

Fix Polyspace Compilation Errors Related to Texas Instruments Compilers

Issue

The Texas Instruments compilers impose a nonstandard requirement on folder sequences in the include file search path, which is not directly supported by the Polyspace project creation mechanism from build systems (using the `polyspace-configure` command).

As a result, if you create a Polyspace project from a build that uses a Texas Instruments compiler, after starting an analysis on the project, you might see the warning:

```
warning: could not find include file "stddef.h"
|  #include_next <stddef.h>
|
```

Possible Solutions

The Texas Instruments compilers impose a certain order of include folders in the include file search path. In particular, the compilers require the implicitly specified `libcxx` subfolder of the compiler include folder to *precede* explicit subfolders in the include file search path. When you create a Polyspace project or options file by tracing a build command, the project or options file contains an include folder sequence where the implicitly included `libcxx` subfolder *comes after* explicit subfolders. As a result, `include_next` lines in files in the `libcxx` subfolder, which only use later include folders in the search path for include file lookup, fail to find the included files.

To work around the problem:

1. In your Polyspace project or options file, locate the `-I` options that point to `libcxx` subfolders. They will appear in lines starting with `-options-for-sources`, for instance:

   ```
   -options-for-sources sourcefile.c;-I some_explicitly_included_folder;-I
   compiler_include_folder\libcxx;
   ```

2. For each such `-options-for-sources` line, switch the order of the `-I`-s so that the `libcxx` subfolder appears first, for instance:

   ```
   -options-for-sources sourcefile.c;-I compiler_include_folder\libcxx;-I
   some_explicitly_included_folder;
   ```

See Also

`polyspace-configure` | `Texas Instruments Compiler (-compiler ti)` | `-options-for-sources` | `-I`

Fix Polyspace Compilation Errors About In-Class Initialization (C++)

When a data member of a class is declared `static` in the class definition, it is a *static member* of the class. You must initialize static data members outside the class because they exist even when no instance of the class has been created.

```
class Test
{
public:

    static int m_number = 0;
};
```

Error message:

```
Error: a member with an in-class initializer must be const
```

Corrected code:

<table>
<tr><th>in file Test.h</th><th>in file Test.cpp</th></tr>
<tr><td><code>class Test
{
public:
static int m_number;
};</code></td><td><code>int Test::m_number = 0;</code></td></tr>
</table>

Fix Polyspace Linking Errors About Conflicting Declarations in Different Translation Units

Issue

The analysis shows an error or warning similar to one of these error messages:

- `Declaration of [...] is incompatible with a declaration in another translation unit ([...])`

  This message appears when the conflicting declarations do not come from the same header file.
- When one of the conflicting declarations is in a header file.

  `Declaration of [...] had a different meaning during compilation of [...] ([...])`

  This message appears when the conflicting declarations come from the same header file included in different source files.

The error indicates that the same variable or function or data type is declared differently in different translation units. The conflicting declarations violate the One Definition Rule (cf. C++Standard, ISO/IEC 14882:2003, Section 3.2). When conflicting declarations occur, Polyspace Code Prover does not choose a declaration and continue analysis.

Common compilation toolchains often do not store data type information during the linking process. The conflicting declarations do not cause errors with your compiler. Polyspace Code Prover follows stricter standards for linking to guarantee the absence of certain run-time errors.

To identify the root cause of the error:

1. From the error message, identify the two source files with the conflicting declarations.

   For instance, an error message looks like this message:

   ```
   C:\field.h, line 1: declaration of class "a_struct" had
         a  different meaning during compilation of "file1.cpp"
   | struct a_struct {
   |
   | Detected during compilation of secondary translation unit "file2.cpp"
   ```

   The message shows that the structure `a_struct` has a conflicting declaration in `file1.cpp` and `file2.cpp`, both of which include the header file `field.h`.

   An alternative error message can look like this:

   ```
   C:\field2.h, line 1: declaration of class "a_struct" had
         is incompatible with a declaration in another translation unit
   | the other declaration is at line 1 of field1.h"
   | struct a_struct {
   |
   | Detected during compilation of secondary translation unit "file2.cpp"
   ```

   The message shows that the structure `a_struct` has a conflicting declaration in `field2.h` and `field.h`. The header file `field2.h` is included in the source file `file2.cpp`.
2. Try to identify the conflicting declarations in the source files.

Otherwise, open the translation units containing these files. Sometimes, the translation units or preprocessed files show the conflicting declarations more clearly than the source files because the preprocessor directives, such as `#include` and `#define` statements, are replaced appropriately and the macros are expanded.

a. Rerun the analysis with the flag `-keep-relaunch-files` so that all translation units are saved. In the user interface, enter the flag for the option `Other`.

   The analysis stops after compilation. The translation units or preprocessed files are stored in a zipped file `ci.zip` in a subfolder `.relaunch` of the results folder.

b. Unzip the contents of `ci.zip`.

   The preprocessed files have the same name as the source files. For instance, the preprocessed file with `file1.cpp` is named `file1.ci`.

When you open the preprocessed files at the line numbers stated in the error message, you can spot the conflicting declarations.

Possible Cause: Variable Declaration and Definition Mismatch

A variable declaration does not match its definition. For instance:

- The declaration and definition use different data types.
- The variable is declared as signed, but defined as unsigned.
- The declaration and definition uses different type qualifiers.
- The variable is declared as an array, but defined as a non-array variable.
- For an array variable, the declaration and definition use different array sizes.

In this example, the code shows a linking error because of a mismatch in type qualifiers. The declaration in `file1.c` does not use type qualifiers, but the definition in `file2.c` uses the `volatile` qualifier.

file1.c	file2.c

`extern int x;`
 	
`void main(void)`
`{/* Variable x used */}`
	`volatile int x;`

In these cases, you can typically spot the difference by looking at the source files. You do not need to see the preprocessed files.

Solution

Make sure that the variable declaration matches its definition.

Possible Cause: Function Declaration and Definition Mismatch

A function declaration does not match its definition. For instance:

- The declaration and definition use different data types for arguments or return values.
- The declaration and definition use a different number of arguments.

- A variable-argument or varargs function is declared in one function, but it is called in another function without a previous declaration.

  In this case, the error message states that the required prototype for the function is missing.

In this example, the code shows a linking error because of a mismatch in the return type. The declaration in `file1.c` has return type `int`, but the definition in `file2.c` has return type `float`.

<table>
<tr><th>file1.c</th><th>file2.c</th></tr>
<tr><td>

```
int input(void);

void main() {
  int val = input();
}
```

</td><td>

```
float input(void) {
  float x = 1.0;
  return x;
}
```

</td></tr>
</table>

In these cases, you can typically find the difference by looking at the source files. You do not need to see the preprocessed files.

Solution

Make sure that the function declaration matches its definition.

Even if your build process allows these errors, you can have unexpected results during run time. If a function declaration and definition with conflicting prototypes exist in your code, when you call the function, the result can be unexpected.

For a variable-argument or varargs function, declare the function before you call it. If you do not want to change your source code, you can work around this linking error.

1. Add the function declaration in a separate file.
2. Only for the purposes of verification, `#include` this file in every source file by using the option `Include (-include)`.

Possible Cause: Conflicts from Unrelated Declarations

You use the same identifier name for two unrelated objects. These are some common reasons for unrelated objects in the same Polyspace project:

- You intended to declare the objects `static` so that they do not have external linkage, but omitted the `static` specifier.
- You declared the same object in several source files instead of putting the declaration in a header file and including in the source files.
- You created a Polyspace project from a build command using the `polyspace-configure` command. The build command created several independent binaries, but files involved in all the binaries were collected in one Polyspace project.

Solution

Depending on the root cause for unrelated objects using the same name, use an appropriate solution.

If your Polyspace project was created from a build command and source files for independent binaries were clubbed together, split the project into modules when tracing your build command. See:

- "Modularize Polyspace Analysis by Using Build Command" on page 4-5
- `polyspace-configure`

Possible Cause: Macro-dependent Definitions

A variable definition is dependent on a macro being defined earlier. One source file defines the macro while another does not, causing conflicts in variable definitions.

In this example, `file1.cpp` and `file2.cpp` include a header file `field.h`. The header file defines a structure `a_struct` that is dependent on a macro definition. Only one of the two files, `file2.cpp`, defines the macro `DEBUG`. The definition of `a_struct` in the translation unit with `file1.cpp` differs from the definition in the unit with `file2.cpp`.

file1.cpp

```
#include "field.h"

int main()
{
    a_struct s;
    init_a_struct(&s);
    return 0;
}
```

file2.cpp

```
#define DEBUG

#include <string.h>
#include "field.h"

void init_a_struct(a_struct* s)
{
    memset(s, 0, sizeof(*s));
}
```

field.h:

```
struct a_struct {
    int n;
#ifdef DEBUG
    int debug;
#endif
};
```

When you open the preprocessed files `file1.ci` and `file2.ci`, you see the conflicting declarations.

file1.ci

```
struct a_struct {
    int n;



};
```

file2.ci

```
struct a_struct {
    int n;

    int debug;

};
```

Solution

Avoid macro-dependent definitions. Otherwise, fix the linking errors. Make sure that the macro is either defined or undefined on all paths that contain the variable definition.

Possible Cause: Keyword Redefined as Macro

A keyword is redefined as a macro, but not in all files.

In this example, `bool` is a keyword in `file1.cpp`, but it is redefined as a macro in `file2.cpp`.

<table>
<tr><th>file1.cpp</th><th>file2.cpp</th></tr>
<tr><td><pre>#include "bool.h"

int main()
{
    return 0;
}</pre></td><td><pre>#define false 0
#define true (!false)

#include "bool.h"</pre></td></tr>
<tr><td colspan="2">bool.h:<pre>template <class T>
struct a_struct {
    bool flag;
    T t;
    a_struct() {
        flag = true;
    }
};</pre></td></tr>
</table>

Solution

Be consistent with your keyword usage throughout the program. Use the keyword defined in a standard library header or use your redefined version.

Possible Cause: Differences in Structure Packing

A `#pragma pack(n)` statement changes the structure packing alignment, but not in all files. See also “Code Prover Assumptions About #pragma Directives”.

In this example, the default packing alignment is used in `file1.cpp`, but a `#pragma pack(1)` statement enforces a packing alignment of 1 byte in `file2.cpp`.

<table>
<tr><th>file1.cpp</th><th>file2.cpp</th></tr>
<tr><td><pre>int main()
{
    return 0;
}</pre></td><td><pre>#pragma pack(1)

#include "pack.h"</pre></td></tr>
<tr><td colspan="2">pack.h:<pre>struct a_struct {
    char ch;
    short sh;
};</pre></td></tr>
</table>

Solution

Enter the `#pragma pack(n)` statement in the header file so that it applies to all source files that include the header.

Fix Errors from Use of Polyspace Header Files

Issue

When analyzing your C/C++ source code with Polyspace, if you do not provide the paths to your compiler headers, Polyspace uses its own version of the headers for the analysis.

In some cases, you might see compilation errors from these Polyspace headers. The error messages typically refer to one of the subfolders of `*polyspaceroot*\polyspace\verifier\cxx\include`. Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2023a`. Typically, the error message is related to a standard library function.

For instance, you might see an error on `std::is_empty` when analyzing this C++14 code:

```
#include <type_traits>

struct S final { };

bool f(void) {
    return std::is_empty<S>::value;
}
```

The error message states:

```
Error: a "final" class type cannot be used as a base class
```

And points to the path *`polyspaceroot`*`\polyspace\verifier\cxx\include\include-libcxx\type_traits`. This error happens because the implementation of `std::is_empty` in Polyspace header files in some cases do not allow their instantiations to use `final` classes.

Possible Solutions

Specify the folders containing *your compiler header files* for the Polyspace analysis.

If you create a Polyspace project or options file from your build command using `polyspace-configure`, the compiler header paths are automatically added to this project or options file. Otherwise, you have to explicitly add these paths:

- In the user interface, add the folders to your project.

  For more information, see "Add Source Files for Analysis in Polyspace Desktop User Interface" on page 2-2.
- At the command line, use the flag `-I` with the `polyspace-code-prover` or `polyspace-code-prover-server` command.

  For more information, see `-I`.

For compilation with GNU C on UNIX-based platforms, use `/usr/include`. On embedded compilers, the header files are typically in a subfolder of the compiler installation folder. Examples of include folders are given for some compilers.

- Wind River Diab: For instance, `/apps/WindRiver/Diab/5.9.4/diab/5.9.4.8/include/`.
- IAR Embedded Workbench: For instance, `C:\Program Files\IAR Systems\Embedded Workbench 7.5\arm\inc`.

- Microsoft Visual Studio: For instance, `C:\Program Files\Microsoft Visual Studio 14.0\VC\include`.

Consult your compiler documentation for the path to your compiler header files. Alternatively, see “Provide Standard Library Headers for Polyspace Analysis” on page 13-19.

See Also

Related Examples

- “Provide Standard Library Headers for Polyspace Analysis” on page 13-19

Fix Polyspace Linking Errors Related to extern "C" Blocks

Extern C Functions

Some functions may be declared inside an `extern "C" { }` block in some files, but not in others. In this case, the linkage is different which causes a link error, because it is forbidden by the ANSI standard.

Original code:

```
extern "C" {
    void* memcpy(void*, void*, int);
}
class Copy
{
public:
    Copy() {};
    static void* make(char*, char*, int);
};
void* Copy::make(char* dest, char* src, int size)
{
    return memcpy(dest, src, size);
}
```

Error message:

```
Pre-linking C++ sources ...

<results_dir>/test.cpp, line 2: error: declaration of function "memcpy"
is incompatible with a declaration in another translation unit
(parameters do not match)
|            the other declaration is at line 4096 of "__polyspace__stdstubs.c"
|    void* memcpy(void*, void*, int);
|          ^
|          detected during compilation of secondary translation unit "test.cpp"
```

The function `memcpy` is declared as an external "C" function and as a C++ function. It causes a link problem. Indeed, function management behavior differs whether it relates to a C or a C++ function.

When such error happens, the solution is to homogenize declarations, i.e. add `extern "C" { }` around previous listed C functions.

Another solution consists in using the permissive option `-no-extern-C`. It removes all `extern "C"` declarations.

Functional Limitations on Some Stubbed Standard ANSI Functions

- `signal.h` is stubbed with functional limitations: `signal` and `raise` functions do not follow the associated functional model. Even if the function raise is called, the stored function pointer associated to the signal number is not called.
- No jump is performed even if the `setjmp` and `longjmp` functions are called.
- `errno.h` is partially stubbed. Some math functions do not set `errno`, but instead, generate a red error when a range or domain error occurs with **ASRT** checks.

You can also use the compile option `POLYSPACE_STRICT_ANSI_STANDARD_STUBS` (`-D` flag). This option only deactivates extensions to ANSI C standard libC, including the functions `bzero`, `bcopy`,

`bcmp`, `chdir`, `chown`, `close`, `fchown`, `fork`, `fsync`, `getlogin`, `getuid`, `geteuid`, `getgid`, `lchown`, `link`, `pipe`, `read`, `pread`, `resolvepath`, `setuid`, `setegid`, `seteuid`, `setgid`, `sleep`, `sync`, `symlink`, `ttyname`, `unlink`, `vfork`, `write`, `pwrite`, `open`, `creat`, `sigsetjmp`, `__sigsetjmp`, and `siglongjmpare`.

Fix Polyspace Compilation Errors About Namespace std Without Prefix

Issue

The Polyspace analysis stops with an error message such as:

```
error: the global scope has no "modfl"
```

The line highlighted in the error uses a function from the standard library without the `std::` prefix.

Cause

Some compilers allow using members of the standard library namespace without explicitly specifying the `std::` prefix. For such compilers, your code can contain lines like this:

```
using ::mblen;
```

where `mblen` is a member of the C++ standard library. Polyspace compilation considers the members as part of the global namespace and shows an error.

Solution

It is a good practice to qualify members of the standard library with the `std::` prefix. For instance, to use the `mblen` function in the preceding example, rewrite the line as:

```
using std::mblen;
```

To continue to retain the current code and work around the Polyspace error, use the analysis option `-using-std`. If you are running the analysis in the Polyspace user interface, enter the option in the **Other** field. See `Other`.

Fix Polyspace Compilation Warnings Related to Assertion or Memory Allocation Functions

Issue

Polyspace uses its own implementation of standard library functions for more efficient analysis. If you redefine a standard library function and provide the function body to Polyspace, the analysis uses your definition.

However, for certain standard library functions, Polyspace continues to use its own implementations, even if you redefine the function and provide the function body. The functions include `assert` and memory allocation functions such as `malloc`, `calloc` and `alloca`.

You see a warning message like the following:

```
Body of routine "malloc" was discarded.
```

Cause

These functions have special meaning for the Polyspace analysis, so you are not allowed to redefine them. For instance:

- The Polyspace implementation of the `malloc` function allows the software to check if memory allocated using `malloc` is freed later.
- The Polyspace implementation of `assert` is used internally to enhance analysis.

Solution

Unless you particularly want your own redefinitions to be used, ignore the warning. The analysis results are based on Polyspace implementations of the standard library function, which follow the original function specifications.

If you want your own redefinitions to be used and you are sure that your redefined function behaves the same as the original function, rename the functions. You can rename the function only for the purposes of analysis using the option `Preprocessor definitions (-D)`. For instance, to rename a function `malloc` to `my_malloc`, use `malloc=my_malloc` for the option argument.

Troubleshoot Java Incompatibility in Polyspace Plugin for Eclipse

Issue

Using the Polyspace desktop plugin for Eclipse and Eclipse-based IDEs require a Java version between 7 and 15. If your Java version is outside this range, after installing the plugin, you might see one of these error messages:

- `Java 7 required, but the current java version is 1.6.`
  `You must install Java 7 before using Polyspace plug in.`
- `Java version 16 is not supported by the Polyspace plugin`

These messages indicate that you have an incompatible version of Java.

Possible Solutions

Check If Eclipse Uses Correct Java Installation

The Polyspace plugin for Eclipse and Eclipse-based IDEs requires a Java version between 7 and 15. If you have installed a compatible Java version, check if your Eclipse IDE is using that Java version:

1. In the Eclipse IDE, click **Help** > **About Eclipse IDE** > **Installation Details**.
2. In the Installation Detail window, on the **Configuration** tab, locate the line `java.version`. This line shows the Java version that is used by the IDE.

If the Java version is incompatible:

1. Install a Java version between 7 and 15, say, in the folder *`java_install`*. Polyspace comes with a compatible Java version in certain platforms. You might prefer to use this version of Java.
2. Open the *`executable_name`*`.ini` file from the Eclipse installation folder.

   For the core Eclipse IDE, the file is `eclipse.ini`. For other Eclipse-based IDEs, the file name might be different.
3. In the file, before the line `-vmargs`, enter:

   ```
   -vm
   java_install\bin\javaw.exe
   ```

   Here, *`java_install`* is the installation folder of the compatible Java version.

   If you prefer to use the Java version Polyspace provides, then enter:

   ```
   -vm
   polyspaceroot\sys\java\jre\arch\jre\bin\javaw.exe
   ```

   Here, *`polyspaceroot`* is your product installation folder, for instance, `C:\Program Files\Polyspace\R2019a\` and *`arch`* is `win32` or `win64` depending on the product platform. Note that `-vm` and the path to `javaw.exe` must be on separate lines.

Switch to Polyspace as You Code

If installing a compatible version of Java is not feasible, consider switching to the Polyspace as You Code plugin for Eclipse. You require a Polyspace Access license to install this plugin. For more information, see “Install Polyspace as You Code Plugin in Eclipse”.

Polyspace as You Code performs a Polyspace Bug Finder analysis on the source file currently open in your IDE. The plugin does not support checking for run-time errors and calculating stack usage by using Polyspace Code Prover. See also:

- “Analysis Scope of Polyspace as You Code”.
- “Checkers Deactivated in Polyspace as You Code Analysis”.

See Also

More About

- “Configure Polyspace as You Code Plugin in Eclipse”
- “Install Polyspace Desktop Plugin for Eclipse”
- “Install Polyspace as You Code Plugin in Eclipse”
- “Checkers Deactivated in Polyspace as You Code Analysis”

Fix Issues When when Integrating Polyspace with MATLAB and Simulink

Issue

Before using Polyspace from MATLAB and Simulink, perform a one-time setup to integrate the two products. For details, see “Integrate Polyspace with MATLAB and Simulink” on page 5-2. When performing the integration steps, if you do not have administrator privileges in MATLAB or your Polyspace installation is nonstandard, you might run into some issues. For instance, the `polyspacesetup()` command might fail to complete.

Possible Solutions

Check if Polyspace installation is nonstandard

By default, Polyspace is installed on the path `C:\Program Files\Polyspace\R2023a`. When you use this command:

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

Polyspace assumes that the installation folder is the default folder. If you install Polyspace in a different folder, the preceding command fails. For more information about the default installation folder, see “Product Installation”.

To resolve this issue, specify the installation path in the `polyspacesetup` command. For instance, at the command line, enter:

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

where *`Folder`* is the Polyspace installation folder.

Check if MATLAB instance has administrator privilege

Executing the command `polyspacesetup` requires administrator privilege. If you do not open MATLAB by using administrator privilege, the command exits with an error.

To resolve this issue, restart MATLAB by using administrator privileges. For instance, in Windows, to open MATLAB with administrator privilege, right-click the MATLAB executable and select **Run as administrator**. In some operating systems, you might need to use an administrator account.

Check If polyspacesetup expects user Input

By default, the `polyspacesetup` command is interactive and expects user input during the integration process. When performing a noninteractive integration, make sure the process is not stuck waiting for your input. To perform a noninteractive installation, at the MATLAB command prompt, enter:

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

where *`Folder`* is the installation path.

Check if Polyspace version is supported

Polyspace integrates completely with MATLAB or Simulink from the same release. If your Polyspace and MATLAB are from different releases, you might not be able to perform a complete integration. See “Polyspace Support of MATLAB and Simulink from Different Releases” on page 6-68.

Depending on your versions of Polyspace and MATLAB, you might be able to partially integrate these products. See “MATLAB Release Earlier Than Polyspace” on page 5-3.

See Also

`polyspacesetup`

More About

- “Polyspace Support of MATLAB and Simulink from Different Releases” on page 6-68
- “Code Prover Analysis with MATLAB Scripts”
- “Code Prover Analysis in Simulink”
- “Code Prover Analysis in MATLAB Coder”
- “Integrate Polyspace with MATLAB and Simulink” on page 5-2

Check Why Polyspace Functions are Unavailable in MATLAB

Issue

To use Polyspace directly from MATLAB or Simulink, you must include the folders containing the Polyspace functions on the MATLAB search path. If these locations are absent or deleted from the search path, the Polyspace functions become unavailable in the MATLAB Command Window.

Possible Solution

Possible Solution: Check if you integrated Polyspace with MATLAB and Simulink

The Polyspace functions are included with a Polyspace installation and their locations are unknown to MATLAB. After installing MATLAB and Polyspace, you cannot use these functions unless you add their locations to the MATLAB search path.

To use the Polyspace functions from MATLAB and Simulink, add their locations to the MATLAB search path by calling the function `polyspacesetup`. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

Possible Solution: Check if Polyspace is supported after a MATLAB update

Updating MATLAB restores the default MATLAB search path and removes any path to the Polyspace installation that was previously added. Because the Polyspace function paths are removed, the integration between Polyspace and MATLAB breaks after a MATLAB update. You must re-establish the integration.

After an update, repeat the steps to integrate Polyspace and MATLAB. If you updated MATLAB to a later release without updating Polyspace, the Polyspace support of MATLAB might be different from before. See “Polyspace Support of MATLAB and Simulink from Different Releases” on page 6-68.

Depending on your versions of Polyspace and MATLAB, integrate these products completely or partially. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

See Also

`polyspacesetup`

More About

- “Polyspace Support of MATLAB and Simulink from Different Releases” on page 6-68
- “Code Prover Analysis with MATLAB Scripts”
- “Code Prover Analysis in Simulink”
- “Code Prover Analysis in MATLAB Coder”
- “Integrate Polyspace with MATLAB and Simulink” on page 5-2

Fix MATLAB Crashes Referring to Polyspace in matlabroot

Issue

In your Polyspace installation, you can find a MATLAB executable in the `*polyspaceroot*\bin` subfolder. The reason is that some functionalities of Polyspace use the MATLAB engine underneath. However, the MATLAB engine shipped with Polyspace is severely reduced and cannot be used by end-users even after license activation.

If you try to open `matlab.exe` from a Polyspace installation folder and try to execute MATLAB commands, MATLAB might crash during command execution. The crash log shows that you opened MATLAB from a Polyspace installation folder, such as `C:\Program Files\Polyspace\R2023a`.

Possible Solutions

Do not open MATLAB from a Polyspace installation by running an executable such as:

```
C:\Program Files\Polyspace\R2023a\bin\matlab.exe
```

Instead, open MATLAB from an actual MATLAB installation by running an executable such as:

```
C:\Program Files\MATLAB\R2023a\bin\matlab.exe
```

To see which MATLAB installation you are using, at the MATLAB command window, enter:

```
matlabroot
```

This command shows you the root of the MATLAB installation.

Note that you can run Polyspace from a MATLAB command line. But even for this usage, you must open MATLAB from your *MATLAB installation folder* and run some preliminary steps to integrate your MATLAB and Polyspace installation. See “Integrate Polyspace with MATLAB and Simulink” on page 5-2.

See Also

Related Examples

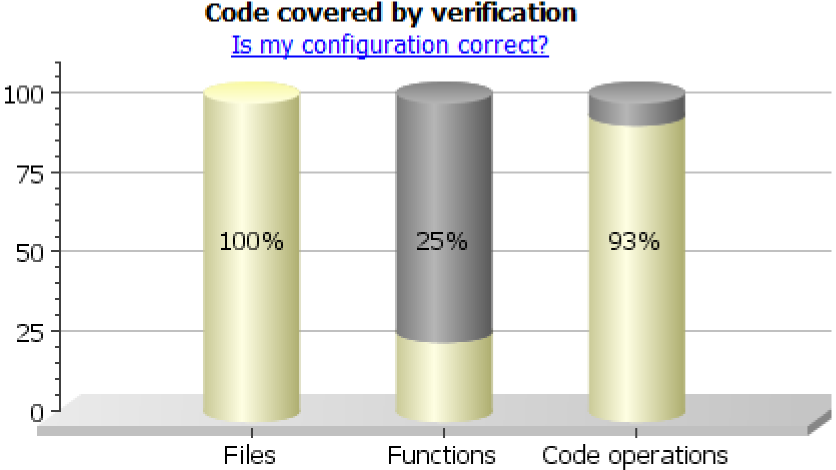
- “Integrate Polyspace with MATLAB and Simulink” on page 5-2
- “Code Prover Analysis with MATLAB Scripts”

Reasons for Unchecked Code

Issue

After verification, you see in the **Code covered by verification** graphs that a significant portion of your code has not been checked for run-time errors.

For instance, in the following graph, the **Dashboard** pane shows that as much as 75% of your functions have not been checked for run-time errors. (In the functions that were checked, only 7% of operations have not been checked.)



The unchecked code percentage in the **Code covered by verification** graph covers:

- Functions and operations that are not checked because they are proven to be unreachable.

  They appear gray on the **Source** pane.

```
    } else {
        *current_data = 200;
    }
```

- Functions and operations that are not proven unreachable but not checked for some other reason.

  They appear black on the **Source** pane.

```
static void proc2(void)
{
    static int SHR3 = 0;

    SHR4.B = 22;
    SHR3 = SHR3 + 1 + SHR4.B + SHR5;
}
```

Click the **Code covered by verification** graph to see a list of unchecked functions.

Possible Cause: Compilation Errors

If some files fail to compile, the Polyspace analysis continues with the remaining files. However, the analysis does not check the uncompiled files for run-time errors.

To see if some files did not compile, check the **Output Summary** or **Dashboard** pane. To make sure that all files compile before analysis, use the option `Stop analysis if a file does not compile (-stop-if-compile-error)`.

Solution

Fix the compilation errors and rerun the analysis.

For more information on:

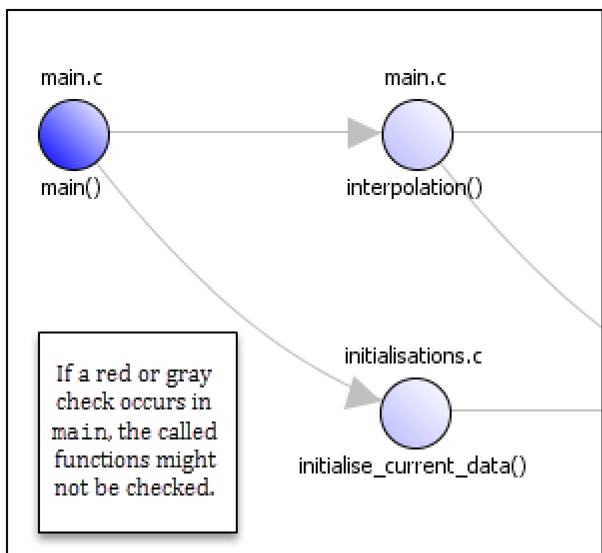
- How the Polyspace compilation works, see “Troubleshoot Compilation and Linking Errors” on page 34-6.
- Specific compilation errors, see “Troubleshoot Compilation Errors”.

Possible Cause: Early Red or Gray Check

You have a red or gray check towards the beginning of the function call hierarchy. Red or grey checks can lead to subsequent unchecked code.

- Red check: The verification does not check subsequent operations in the block of code containing the red check.
- Gray check: Gray checks indicate unreachable code. The verification does not check operations in unreachable code for run-time errors.

If you call functions from the unchecked block of code, the verification does not check those functions either. If you have a red or gray check towards the beginning of the call hierarchy, functions further on in the hierarchy might not be checked. You end up with a significant amount of unchecked code.



For instance, in the following code, only 1 out of 4 functions are checked and the **Procedure** graph shows 25%. The functions `func_called_from_unreachable_1`, `func_called_from_unreachable_2` and `func_called_after_red` are not checked. Only `main` is checked.

```
void func_called_from_unreachable_1(void) {
}

void func_called_from_unreachable_2(void) {
}

void func_called_after_red(void) {
}

int glob_var;

void main(void) {
     int loc_var;
     double res;

     glob_var=0;
     glob_var++;

     if (glob_var!=1) {
           func_called_from_unreachable_1();
           func_called_from_unreachable_2();
     }

     res=0;
     /* Division by zero occurs in for loop */
     for(loc_var=-10;loc_var<10;loc_var++) {
              res += 1/loc_var;
     }

     func_called_after_red();
}
```

Solution

See if the `main` function or another Tasks function has red or gray checks. See if you call most of your functions from the subsequent unchecked code.

To navigate from the `main` down the function call hierarchy and identify where the unchecked code begins, use the navigation features on the **Call Hierarchy** pane. If you do not see the pane by default, select **Window** > **Show/Hide View** > **Call Hierarchy**. For more information, see "Call Hierarchy in Polyspace Desktop User Interface" on page 21-24.

Alternatively, you can consider an arbitrary unchecked function and investigate why it is not checked. See if the same reasoning applies for many functions. To detect if a function is not called at all from an entry point or called from unreachable code, use the option `Detect uncalled functions (-uncalled-function-checks)`.

Review the red or gray checks and fix them.

Possible Cause: Incorrect Options

You did not specify the necessary analysis options. When incorrectly specified, the following options can cause unchecked code:

- Multitasking options: If you are verifying multitasking code, through these options, you specify your entry point functions.

  Possible errors in specification include:

  - You expected automatic concurrency detection to detect your thread creation, but you use thread creation primitives that are not yet supported for automatic detection.
  - With manual multitasking setup, you did not specify all your entry points.
- Main generation options: Through these options, you generate a `main` function if it does not exist in your code. When verifying modules or libraries, you use these options.

  You did not specify all the functions that the generated `main` must call.
- Inputs and stubbing options: Through these options, you constrain variable ranges from outside your code or force stubbing of functions.

  Possible errors in specification include:

  - You specified variable ranges that are too narrow causing otherwise reachable code to become unreachable.
  - You stubbed some functions unintentionally.
- "Macros": Through these options, you define or undefine preprocessor macros.

  You might have to explicitly define a macro that your compiler considers implicitly as defined.

Solution

Check your options in the preceding order. If your specifications are incorrect, fix them.

Possible Cause: main Function Does Not Terminate

This cause applies only for multitasking code when entire entry-point functions are not checked.

If you configure multitasking options manually, you must follow the restrictions on the Polyspace multitasking model. In particular, the `main` function must not contain an infinite loop or a run-time error. Otherwise, entry-point functions are not checked.

For instance, in this example, `task2` is not checked even if you specify it as an entry point. The reason is the infinite loop in the `main` function.

```
void performTask1Cycle(void);
void performTask2Cycle(void);

void main() {
 while(1) {
    performTask1Cycle();
  }
}

void task2() {
 while(1) {
    performTask2Cycle();
  }
}
```

You see the `while` keyword in the `main` function underlined in dashed red. A tooltip indicates that the loop might not terminate.

Likewise, if a run-time error occurs, the function call leading to the run-time error is underlined in dashed red with an accompanying tooltip.

Solution: Terminate main Function

Fix the reason why the `main` function does not terminate.

- If the reason is a definite run-time error (red check), fix the error.
- If the reason is an infinite loop, see why the loop must be infinite.

  If the infinite loop in the `main` function represents a cyclic task, terminate the `main` function and move the infinite loop to another entry-point function. You can make this change only for the purposes of Polyspace analysis without actually modifying your `main` function. See "Configuring Polyspace Multitasking Analysis Manually" on page 15-17.

Identify Why Some Files or Functions are Missing in Polyspace Results

In this section...
"Issue" on page 34-83
"Possible Cause: Files Not Verified" on page 34-83
"Possible Cause: Filters Applied" on page 34-84

Issue

On the **Results List** pane, when you select **File** from the (Grouping) list, you do not see:

- Some of your source files.
- Some functions in your source files.

Possible Cause: Files Not Verified

If a source file or function does not contain a result such as a check or coding rule violation, the **Results List** pane does not display the file or function. If none of the operations in a source file or function contain a check, it indicates that Polyspace did not verify that source file or function.

To check if all files and functions were verified, see the **Code covered by verification** graph on the **Dashboard** pane. For more information, see "Dashboard in Polyspace Desktop User Interface" on page 21-7.

Solution

Polyspace does not verify a source file or function when one of the following situations occur.

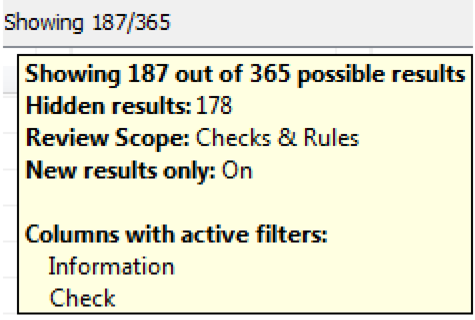
Situation	Fix
The file or function does not contain an operation on which a check is required. For instance, a function contains calls to other functions only. If none of the called functions contains an error that lead to a Non-terminating call error in the calling function, the calling function does not contain a check.	No fix required.
All functions in the source file are not called, are called from unreachable code or are called following red checks. Polyspace does not verify the code that follows a red check and occurs in the same scope as the check. Therefore, it considers that the functions are not called and does not verify the file containing the functions.	If you choose to detect uncalled functions, the verification places a gray check on those functions. The functions and the source file containing the functions then appear on the **Results List** pane. For more information, see `Detect uncalled functions (-uncalled-function-checks)`.

Situation	Fix
Your code is intended for multitasking and you do not specify all your entry points. If all functions in a file are called from an entry point function that you did not specify, Polyspace does not verify the file.	See if you specified all entry points. For more information on how to specify entry points, see `Tasks (-entry-points)`. For a workflow on verifying multitasking code, see "Configuring Polyspace Multitasking Analysis Manually" on page 15-17.
If your source files do not contain a `main` function, Polyspace generates a `main` function. The generated `main` calls the functions that you specify using certain analysis options. If your analysis options are such that the generated `main` does not call all the functions in a source file, Polyspace does not verify the source file.	See if you have to change the `main` generation options associated with your verification. For more information on the options, see: • `Initialization functions (-functions-called-before-main)` • `Functions to call (-main-generator-calls)` • `Class (-class-analyzer)` • `Functions to call within the specified classes (-class-analyzer-calls)`.

Possible Cause: Filters Applied

If you rerun verification on a project module, filters from the last run are applied to the current run. Because of the persistent filters, some of the files can be hidden from display.

To check if some filters are applied, see the **Results List** pane header. The header shows the number of results filtered from the display. If you place your cursor on this number, you can see the applied filters.



For instance, in the image, you can see that the following filters have been applied:

- The **Checks & Rules** filter to suppress code metrics and global variables.
- The New filter to suppress results found in a previous verification.
- Filters on the **Information** and **Check** columns.

Solution

Clear the filters and see if your file or function reappears on the **Results List** pane. For more information, see “Filter and Group Results in Polyspace Desktop User Interface” on page 24-2.

Fix Polyspace Overapproximations on Standard Library Math Functions

Issue

In your verification results, a standard library math function does not behave as expected.

For instance, the statement `assert(isinf(x))` does not constrain the value of `x` to positive or negative infinity in subsequent statements.

Cause

If Polyspace cannot find the math function definitions, the verification uses Polyspace implementations of the standard library math functions.

In some cases, the Polyspace implementation of the function might not match the function specification. Note that in such cases, the Polyspace implementation overapproximates the function behavior. For instance, following the statement `assert(isinf(x))`, the range of values of `x` include positive and negative infinity. Therefore, such behavior does not lead to green checks for operations that can cause run-time errors.

Solution

Explicitly provide the path to your compiler's native header files so that the verification uses your compiler's implementations of the functions. For instance, some compilers implement functions such as `isinf` as macros in their header files.

- If you are running verification from the command line, use the option `-I`.
- If you are running verification from the user interface, see "Add Source Files for Analysis in Polyspace Desktop User Interface" on page 2-2.

If you use a cross compiler and create a Polyspace project from your build system, the project uses the header files provided by your compiler.

Avoid Red Checks in Unreachable Code When Using C++ STL Containers

Issue

By default, Code Prover analyzes the implementations of methods in STL containers such as `std::map`, `std::vector`, and `std::list`. This analysis can be time-consuming and sometimes lead to issues such as red checks in otherwise unreachable code.

For instance, you might see issues such as a red check in an `if` statement branch, even if the `if` condition is always false in the current context (and the code in the `if` statement can never execute or trigger a run-time error).

Possible Solutions

Enable a Code Prover analysis mode that does not analyze implementations of methods in STL containers. Specify the value `stdlibcxx` for the option `Libraries used (-library)`.

In this mode, the analysis uses smart stubs for methods from C++ Standard Library containers even if their implementations are available. The smart stubs emulate STL container methods only as much as needed for the verification. In exchange, the stubs work around the problem of long analysis times or red checks in unreachable code. For the full list of containers that are supported with smart stubs, see `Libraries used (-library)`.

Note that all methods of a container might not be stubbed when you use this option. A message in the analysis log states how many methods were stubbed. With each release of Polyspace Code Prover, more container methods are covered by the smart stubbing.

See Also

`Libraries used (-library)`

Fix Insufficient Memory Errors During Polyspace Report Generation

Issue

When generating reports from Polyspace results containing a very large number of defects or coding rule violations, you might encounter insufficient memory errors.

The error message can look like this message:

```
....
Exporting views...
Initializing...
Polyspace Report Generator
Generating Report
 .....
    Converting report
Opening log file:  C:\Users\auser\AppData\Local\Temp\java.log.7512
Document conversion failed
.....
Java exception occurred:
java.lang.OutOfMemoryError: Java heap space
```

Possible Solutions

To resolve the issue, you can try increasing the available heap memory or reporting the results over multiple reports instead of in a single report.

Increase Java Heap Size

If the error occurs during report generation, try increasing the Java heap size. The default heap size in a 64-bit architecture is 1024 MB.

To increase the size:

1. Navigate to `*polyspaceroot*\polyspace\bin*architecture*`. Where:
   - *`polyspaceroot`* is the installation folder.
   - *`architecture`* is your computer architecture, for instance, `win32`, `win64`, etc.
2. Change the default heap size that is specified in the file, `java.opts`. For example, to increase the heap size to 2 GB, replace `1024m` with `2048m`.
3. If you do not have write permission for the file, copy the file to another location. After you have made your changes, copy the file back to *`polyspaceroot`*`\polyspace\bin\`*`architecture`*`\`.

Report Polyspace Results in Multiple Reports

Instead of reporting all results from a result set (`.pscp` or `.psbf` file) in a single report, you can generate multiple reports, each containing a smaller subset of results.

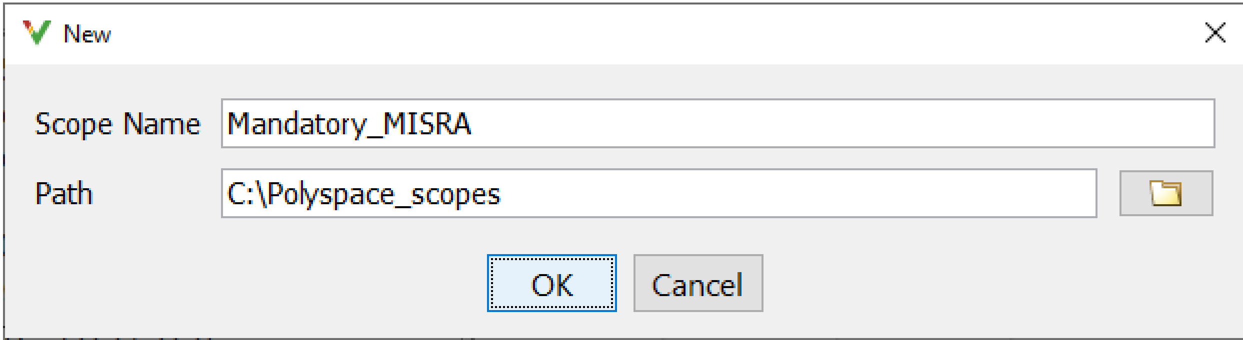
The simplest strategy can be to report results of a certain type in a single report. For instance, from a results set containing MISRA C:2012 rule violations, you can generate two reports, one for the mandatory and required rules, and another for the remaining rules.

Apply Review Scopes and Generate Filtered Report

You can create named sets of filters called review scopes in the user interface of the Polyspace desktop products. When generating a report, you can apply a review scope that filters the results before storing them in the report.

To create a review scope:

1. In the Polyspace user interface, select **Tools > Preferences**.
2. On the **Review Scope** tab, select **New**. Enter a review scope name and a location to save.

   The review scope name can be used later when filtering the results.
3. Select the checkers that you want to be included in the review scope.

You can generate a filtered report from a Polyspace results set by applying a review scope before report generation. The report contains results of only those checkers that are included in the review scope. Note that even though you create the review scope in the user interface, you can apply the scope for report generation in the user interface or at the command line.

To generate a filtered report in the Polyspace user interface:

1. In the dropdown at the top of the **Results List** pane, instead of **All results**, select your new review scope. The list of results is narrowed down to results of only those checkers that are included in the review scope.
2. Generate a report or export results using the **Reporting** menu.

   - To generate reports, select **Run Report**. In the Run Report dialog, select **Only include currently displayed results** before starting report generation.
   - To export results select **Export > Export Currently Displayed Results**.

To generate a filtered report at the command line, use the option `-wysiwyg` *`scopeName`* with the command `polyspace-report-generator`. Here, *`scopeName`* is the review scope name that you used when saving the scope from the Polyspace user interface. For instance:

```
polyspace-report-generator
    -template templateName
    -results-dir resultsFolder
    -format HTML -wysiwyg scopeName
```

Here:

- `templateName` is the path to a report template, such as `polyspaceroot\toolbox\polyspace\psrptgen\templates\Developer.rpt`, where `polyspaceroot` is the Polyspace installation folder such as `C:\Program Files\Polyspace\R2023a`.
- `resultsFolder` is the folder containing Polyspace results.

Note that the scope names are stored with your user preferences. Therefore, you cannot use the scopes names for report generation with another Polyspace installation, for instance, from a different release. For more information on user preferences, see “Storage of Polyspace User Interface Customizations” on page 2-27.

Apply Column-Based Results List Filters and Generate Filtered Report

You can also filter results in the Polyspace user interface using column-based filters and create a filtered report from the currently displayed results:

1. Click the icon on column headers of the **Results List** pane to see the available filters. Apply the filters that you want.
2. Generate a report or export results using the **Reporting** menu.
   - To generate reports, select **Run Report**. In the Run Report dialog, select **Only include currently displayed results** before starting report generation.
   - To export results select **Export** > **Export Currently Displayed Results**.

The generated report states which columns were used to filter results. For instance, if you use the **Status** column to suppress all results with status `Justified`, the generated report contains this line:

```
Columns with active filters:
   Status
```

All results with status `Justified` do not appear in the report.

Your choice of column-based filters are stored in the file `ui_settings.prf` in the `.settings` subfolder of the results folder. Therefore, if you generate a report from these results at the command line and use the option `-wysiwyg "All results"` (or with a scope name as shown in previous section), your choice of column-based filters apply to the generated report. You can even move the file `ui_settings.prf` to `.settings` subfolders of other results folders to generate filtered reports from those other results.

See Also

`polyspace-report-generator`

More About

- “Generate Reports from Polyspace Results” on page 25-2

Fix Polyspace Errors Related to Temporary Files

Polyspace produces some temporary files during analysis. The following issues are related to storage of temporary files.

No Access Rights

When running verification, you get an error message that Polyspace could not create a folder for writing temporary files. For instance, the error message can be as follows:

```
Unable to create folder "C:\Temp\Polyspace\foldername
```

Cause

Polyspace produces some temporary files during analysis. If you do not have write permissions for the folder used to store the files, you can encounter the error.

Solution

There are two possible solutions to this error:

- Change the permissions of your temporary folder so you have full read and write privileges.

  To learn how Polyspace determines the temporary folder location, see "Storage of Temporary Files During Polyspace Analysis" on page 3-6.
- Use the option `-tmp-dir-in-results-dir`. Instead of the standard temporary folder, Polyspace uses a subfolder of the results folder.

No Space Left on Device

When running verification, you get an error message that there is no space on a device.

Cause

If you do not have sufficient space on for the folder used to store the files, you can encounter the error.

Solution

There are two possible solutions to this error:

- Change the temporary folder to a drive that has enough disk space.

  To learn how Polyspace determines the temporary folder location, see "Storage of Temporary Files During Polyspace Analysis" on page 3-6.
- Use the option `-tmp-dir-in-results-dir`. Instead of the standard temporary folder, Polyspace uses a subfolder of the results folder.

Cannot Open Temporary File

When running verification, you get an error message that Polyspace could not open a temporary file.

Cause

You defined the path for storing temporary files by using the environment variable `RTE_TMP_DIR`. You either used a relative path for the temporary folder, the folder does not exist or you do not have access rights to the folder.

Solution

There are two possible solutions to this error:

- Instead of defining a temporary folder specific to Polyspace through `RTE_TMP_DIR`, use a standard temporary folder.

  To learn how Polyspace determines the temporary folder location, see “Storage of Temporary Files During Polyspace Analysis” on page 3-6.
- If you continue to use `RTE_TMP_DIR`, make sure you specify an absolute path to an existing folder and you have access rights to the folder.

Fix Errors or Slow Polyspace Runs from Disk Defragmentation and Anti-virus Software

Issue

In some cases, anti-virus software checks can noticeably slow down a Polyspace analysis. This reduction occurs because the software checks the temporary files produced by the Polyspace analysis.

You see noticeably slow analysis for a simple project or the analysis stops with an error message like the following:

```
Some stats on aliases use:
  Number of alias writes:      22968
  Number of must-alias writes: 3090
  Number of alias reads:       0
  Number of invisibles:        949
Stats about alias writes:
  biggest sets of alias writes: foo1:a (733), foo2:x (728), foo1:b (728)
  procedures that write the biggest sets of aliases: foo1 (2679), foo2 (2266),
                                                       foo3 (1288)
**** C to intermediate language translation - 17 (P_PT) took 44real, 44u + 0s (1.4gc)
exception SysErr(OS.SysErr(name="Directory not empty", syserror=notempty)) raised.
unhandled exception: SysErr: No such file or directory [noent]


--------------------------------------------------------------------------
---                                                                    ---
---  Verifier has encountered an internal error.      ---
---  Please contact your technical support.         ---
---                                                                    ---
---------------------------------------------------------------------------
```

Possible Cause

A disk defragmentation tool or anti-virus software is running on your machine.

After starting an analysis, check the processes running and see if an anti-virus process is causing large amount of CPU usage (and possibly memory usage).

Solution

Try:

- Stopping the disk defragmentation tool.
- Deactivating the anti-virus software. Or, configuring exception rules for the anti-virus software to allow Polyspace to run without a failure.

  For instance, you can try the following:

  - Configure the anti-virus software to allow the Polyspace executables.

    For instance, in Windows, with the anti-virus software Windows Defender, you can add an exclusion for the Polyspace installation folder `C:\Program Files\Polyspace\R2019a`, in particular, the `.exe` files in the subfolder `polyspace\bin` and the `.exe` files starting with `polyspace` in the subfolder `bin\win64` (for instance, `polyspace-internal-connector.exe`).

- Configure the anti-virus software to exclude your temporary folder, for example, `C:\Temp`, from the checking process.

  See “Storage of Temporary Files During Polyspace Analysis”.

Fix SQLite I/O Errors on Running Polyspace

Issue

Polyspace uses an SQLite database for storing analysis results. SQLite databases might show problems when shared on network file systems such as NFS (Network File System), CIFS (Common Internet File System) or SMB (Server Message Block protocol), and the like.

If you save your analysis results on network file systems, you might see errors like this:

```
exception SQLError(SQLite.SQLError(code=10, disk I/O error (errcode=10 extended errcode=1034)))
raised.
```

The errors indicate that the database disk image is malformed and the results are possibly corrupted.

Possible Solutions

Check the folder where you save Polyspace results. For instance, if you run Polyspace at the command line, check the argument of the option `-results-dir`.

If the folder is a network folder that uses file systems such as NFS, use a local folder instead.

Fix License Error –4,0 When Running Polyspace

Issue

When you try to run Polyspace, you get this error message:

```
License Error -4,0
```

Possible Cause: Another Polyspace Instance Running

You can open multiple instances of Polyspace, but you can only run one code analysis at a time.

If you try to run Polyspace processes from multiple windows, you will get a `License Error –4,0` error.

Solution

Only run one analysis at a time, including any command-line or plugin analyses.

Possible Cause: Prior Polyspace Run in Simulink or MATLAB Coder

If you run Polyspace on generated code in the Simulink user interface or in the MATLAB Coder app, you can get a license error if you try to run a subsequent analysis in the Polyspace user interface. You get the error even if the previous run is over.

Solution

Run the subsequent analysis using the method that you used before, that is, in the Simulink user interface or MATLAB Coder app.

If you want to run the analysis in the Polyspace user interface, close Simulink or MATLAB Coder and then rerun the analysis.

Fix Errors Applying Custom Annotation Format for Polyspace Results

Issue

When you use the option `-xml-annotations-description` to apply custom annotations to your Polyspace results, some custom annotations are not applied and you see warnings in the console output or the desktop interface **Output Summary**.

Possible Solutions

Custom Annotation Not Found in Mapping

If you define a custom annotation syntax but you do not map it to the Polyspace annotation syntax, Polyspace detects the custom annotation but does not apply it to the analysis results. You see a warning similar to this warning in the console output or the Polyspace desktop interface **Output Summary**.

```
Verifying sources ...
Verifying zero_div.c (1/1)
Warning: rule :50 from exampleCustomAnnotation not found in the mapping (XML file).
         Skipping the annotation
```

Solution

Check the `<Mapping/>` section of the XML file that you pass to the `-xml-annotations-description` option. If the rule listed in the warning is not mapped to a Polyspace rule, add the appropriate entry to map the rule. For instance, to map rule 50 from the preceding warning to Polyspace coding rule **MISRA C: 2012 Rule 8.4**, add this entry in the `<Mapping/>` section:

```
<Result_Name_Mapping  Rule_Identifier="50" Family="MISRA-C3" Result_Name="8.4"/>
```

Polyspace Annotations Do Not Apply to Current Code

If you define a custom annotation syntax and you map it to the Polyspace annotation syntax, Polyspace might not apply some custom annotations to your source code. You see a warning similar to this warning in the console output or the Polyspace desktop interface **Output Summary**.

```
Warning: These Polyspace annotations do not apply to the current code:
|          In file D:\Polyspace\Examples\zero_div.c line 7, annotation MISRA-C3:8.4 with text
"Justified by annotation in source"
|          In file D:\Polyspace\Examples\zero_div.c line 20, annotation MISRA-C3:8.4 with text
"Justified by annotation in source"
|        Possible reasons:
|          - Issue not detected with selected configuration options.
|          - Issue is fixed.
|          - Annotation syntax is incorrect
```

Solution

Check for these possible causes:

- The issue that the annotation addresses has been fixed in the source code. Polyspace detects the custom annotation but ignores it.
- The issue that the annotation addresses was not detected by Polyspace with the analysis options that you specified. For example, if the custom annotation addresses a MISRA C: 2012 coding standard violation but Polyspace did not check for violations of this coding standard because option `Check MISRA C:2012 (-misra3)` is not specified.

- The issue that the annotation addresses was detected but Polyspace could not match the custom annotation to a corresponding Polyspace annotation. This indicates a syntax error in the `<Mapping/>` section of the XML file that you pass to the `-xml-annotations-description` option.

See Also

`-xml-annotations-description`

Related Examples

- "Define Custom Annotation Format" on page 30-20

Troubleshooting Polyspace Access

Polyspace Access ETL and Web Server services do not start

Issue

You start the Polyspace Access services but after a moment, the **ETL** and **Web Server** services stop. You might see a HTTP 403 error message in your web browser when you try to connect to Polyspace Access.

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

On Windows, if you installed Polyspace Access inside a virtual machine (VM), that VM is managed by Hyper-V. Depending on your network configuration, Hyper-V might not resolve local host names. The **Polyspace Access ETL** and **Polyspace Access Web Server** services cannot connect to the host that you specify with these host names.

To test whether Hyper-V can resolve host name `myHostname` on a machine that is connected to the Internet, at the command line, enter:

```
docker run --rm -it alpine ping myHostname
```

If Hyper-V cannot resolve the host name, you get an error message.

Solution

Stop and restart the `admin-docker-agent` binary without using the `--hostname` option.

- If you are on a trusted network or you do not want to use the HTTPS protocol:

  1. At the command-line, enter:

     ```
     docker stop admin
     ```

     ```
     admin-docker-agent --restart-gateway
     ```
  2. In the **Cluster Admin** web interface, click **Restart Apps**.
- If you want to use the HTTPS protocol, generate certificates with a subject alternative name (SAN) that includes the IP address of the cluster operator node on which the services are running.

  1. Copy this configuration file to a text editor and save it on your machine as `openssl.cnf`.

     Configuration file

     ```
     [ req ]
     req_extensions = v3_req
     distinguished_name = req_distinguished_name
     prompt = no

     [ req_distinguished_name ]
     countryName = US
     stateOrProvinceName = yourState
     organizationName = myCompany
     organizationalUnitName = myOrganization
     emailAddress = user@email.com
     commonName = hostName
     ```

```
[ v3_req ]
basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
subjectAltName = @alt_names

[ alt_names ]
DNS.1 = hostName
DNS.2 = fullyQualifiedDomainName
IP.1 = nodeIPAddress
```

hostName is the host name of the server that hosts Polyspace Access. *fullyQualifiedDomainName* is the corresponding fully qualified domain name. *nodeIPAddress* is the IP address of the node on which you run the `admin-docker-agent` binary.

You do not need to edit the value of the other fields in the `[ req_distinguished_name ]` section of `openssl.cnf`. Updating the value of these fields does not affect the configuration.

2. In the **Cluster Dashboard**, click **Configure Nodes**. The IP address listed in the **Hostname** field corresponds to *nodeIPAddress* in the `openssl.cnf` file. If there is more than one node listed, add an additional line in the `[ alt_names ]` section of `openssl.cnf` for each IP address. For example:

```
[ alt_names ]
DNS.1 = hostName
DNS.2 = fullyQualifiedDomainName
IP.1 = nodeIPAddress
IP.2 = additionalNodeIPAddress
```

3. Generate a certificate signing request (CSR) by using your `openssl.cnf` configuration file. At the command line, enter:

```
openssl req -new -out myReqest.csr -newkey rsa:4096 \
-keyout myKey.key -nodes -config openssl.cnf
```

The command outputs a private key file `myKey.key` and the file `myRequest.csr`.

4. To generate a signed certificate:

   - If you use your organization's certificate authority, submit `myRequest.csr` to the certificate authority. The certificate authority uses the file to generate a signed server certificate. For instance, `server_cert.cer`.
   - If you use self-signed certificates, at the command line, enter:

   ```
   openssl x509 -req -days 365 -in myRequest.csr -signkey myKey.key \
   -out self-cert.pem -extensions v3_req -extfile openssl.cnf
   ```

   The command outputs self-signed certificate `self-cert.pem`.

5. Stop and restart the `admin-docker-agent` binary with this command:

Windows PowerShell	`./admin-docker-agent --restart-gateway ` ` `--ssl-cert-file certFile1 ` ` `--ssl-key-file keyFile ` ` `--ssl-ca-file trustedStoreFile`

Linux	`./admin-docker-agent --restart-gateway \`
	`--ssl-cert-file certFile1 \`
	`--ssl-key-file keyFile \`
	`--ssl-ca-file  trustedStoreFile`

`certFile1` is the full path of the file you obtained in step 4.`keyFile` is the file you generated in step 3. `trustedStoreFile` is the file you generated in step 4 if you used self-signed certificates. Otherwise, it is the trust store file you use to configure HTTPS. See “Choose Between HTTP and HTTPS Configuration for Polyspace Access” Save your changes.

6. In the **Cluster Admin** web interface, click **Restart Apps**.

Contact Technical Support About Polyspace Access Issues

If you need support from MathWorks for Polyspace Access issues, go to this page and create a service request. You need a MathWorks login and password to create a service request.

Before you contact MathWorks, gather this information.

- **Operating system**

  To see information about the operating system of the machine where you install Polyspace access, at the command line, enter:

Windows	`systeminfo | findstr /C:OS`
Linux	`uname -a`

- **Polyspace Access version**

  Log into Polyspace Access, then at the top of the window click ? > **About Polyspace**. If Polyspace Access is not yet installed or you cannot log into Polyspace Access, at the command line, navigate to the folder where you unzipped the Polyspace Access installation image, and enter:

Windows	`type VERSION`
Linux	`cat VERSION`

- **License number**

  Log into Polyspace Access, then at the top of the window click ? > **About Polyspace**. If Polyspace Access is not yet installed or you cannot log into Polyspace Access, contact your license administrator to obtain your license number.

- **Polyspace Access service logs**

  To generate logs for the different Polyspace Access services, at the command line, enter:

```
docker logs -t polyspace-access-web-server-0-main >> access-web-server.log 2>&1
docker logs -t polyspace-access-etl-0-main >> access-etl.log 2>&1
docker logs -t polyspace-access-db-0-main >> access-db.log 2>&1
docker logs -t polyspace-access-download-0-main >> download-service.log 2>&1
docker logs -t issuetracker-server-0-main >> issuetracker-server.log 2>&1
docker logs -t issuetracker-ui-0-main >> issuetracker-ui.log 2>&1
docker logs -t usermanager-server-0-main >> usermanager-server.log 2>&1
docker logs -t admin >> admin.log 2>&1
docker logs -t gateway >> gateway.log 2>&1
docker logs -t usermanager-ui-0-main >> usermanager-ui.log 2>&1
docker logs -t usermanager-db-0-main >> usermanager-db.log 2>&1
docker logs -t polyspace-access >> polyspace-access.log 2>&1
docker logs -t issuetracker >> issuetracker.log 2>&1
docker logs -t usermanager>> usermanager.log 2>&1
```

  Attach the log files to your service request. The commands to generate these logs are the same for Windows and Linux.

  Note If you run Polyspace Access version R2021b or earlier, the docker container names might be different. To view the names of currently running containers, use command `docker ps --format '{{.Names}}'`.

- **Polyspace Access web interface log**

To generate a log for the Polyspace Access web interface, log into Polyspace Access. In the left pane, click **SUPPORT REPORT** then **Get support info**. Attach the generated `supportreport` file to your service request.