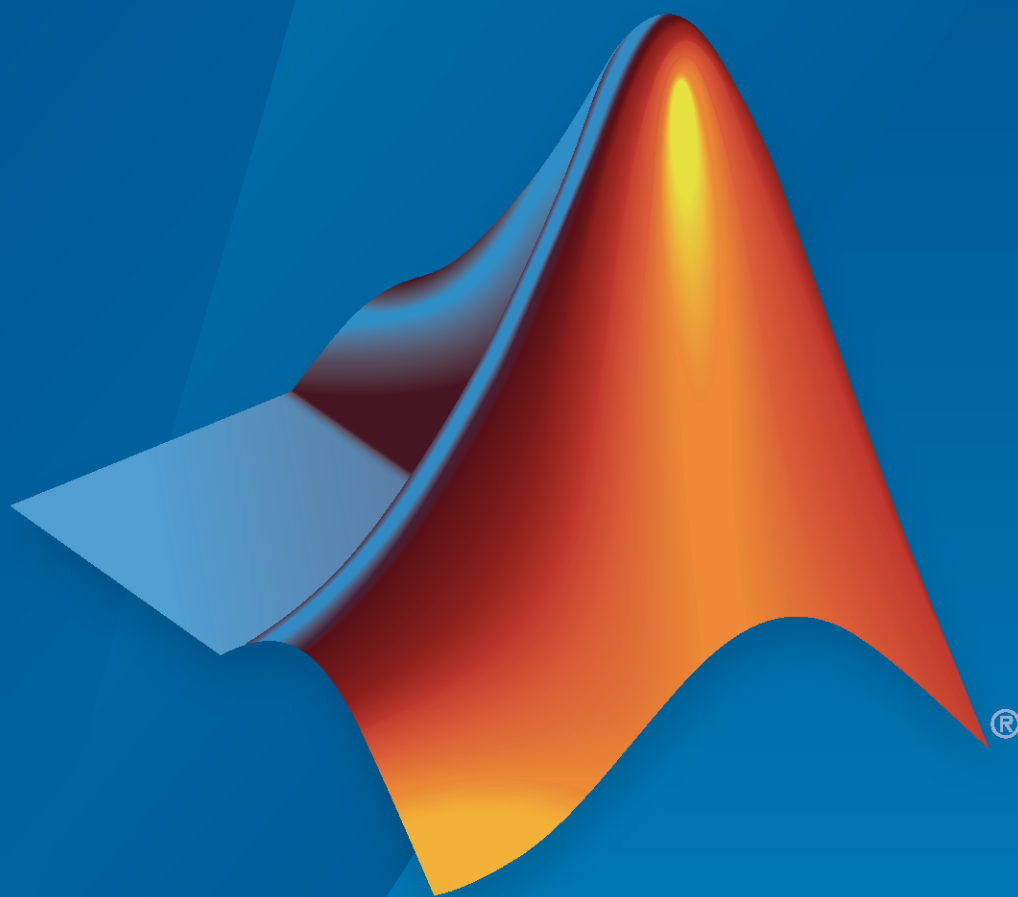


Polyspace® Code Prover™

User's Guide



R2020a



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 2013–2020 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)

1

Run Polyspace Analysis on Desktop

Add Source Files for Analysis in Polyspace User Interface	1-2
Add Sources from Build Command	1-2
Add Sources Manually	1-4
Run Polyspace Analysis on Desktop	1-7
Arrange Layout of Windows for Project Setup	1-7
Set Product and Result Location	1-8
Start and Monitor Analysis	1-9
Fix Compilation Errors	1-9
Open Results	1-10
Project and Results Folder Contents	1-11
Files in the Results Folder	1-11
Storage of Temporary Files	1-12
Create Project Using Visual Studio Information	1-13
Create Project Using Configuration Template	1-15
Why Use Templates	1-15
Use Predefined Template	1-15
Create Your Own Template	1-15
Update Polyspace Project	1-18
Change Folder Path	1-18
Refresh Source List	1-19
Refresh Project Created from Build Command	1-19
Add Source and Include Folders	1-19
Manage Include File Sequence	1-19
Organize Layout of Polyspace User Interface	1-21
Create Your Own Layout	1-21
Save and Reset Layout	1-22
Customize Polyspace User Interface	1-23
Possible Customizations	1-23
Storage of Polyspace User Interface Customizations	1-25
Upload Results to Polyspace Access	1-27
Upload Results from Polyspace Desktop Client	1-27
Upload Results at Command Line	1-28

Run Polyspace Analysis with Windows or Linux Scripts

2

Run Polyspace Analysis from Command Line	2-2
Specify Sources and Analysis Options Directly	2-2
Specify Sources and Analysis Options in Text File	2-2
Create Options File from Build System	2-3
Modularize Polyspace Analysis by Using Build Command	2-4
Build Source Code	2-4
Create One Polyspace Options File for Full Build	2-6
Create Options File for Specific Binary in Build Command	2-7
Create One Options File Per Binary Created in Build Command	2-7
polyspace-configure Source Files Selection Syntax	2-10
Configure Polyspace Analysis Options in User Interface and Generate Scripts	2-12
Prerequisites	2-13
Generate Scripts from Configuration	2-13
Run Analysis with Generated Scripts	2-14

Run Polyspace Analysis with MATLAB Scripts

3

Integrate Polyspace with MATLAB and Simulink	3-2
Integrate Polyspace with MATLAB and Simulink Installation from Same Release	3-2
Integrate Polyspace with Simulink Installation from Different Release	3-3
Check Integration Between MATLAB and Polyspace	3-4
Run Polyspace Analysis by Using MATLAB Scripts	3-5
Prerequisites	3-5
Specify Multiple Source Files	3-5
Check for MISRA C:2012 Violations	3-6
Check for Specific Defects or Coding Rule Violations	3-6
Find Files That Do Not Compile	3-7
Run Analysis on Server	3-7
Compare Results from Different Polyspace Runs by Using MATLAB Scripts	3-9
Review Only New Results Compared to Last Run	3-9
Review New Results and Unreviewed Results from Last Run	3-10
Generate MATLAB Scripts from Polyspace User Interface	3-11
Prerequisites	3-11
Create Scripts from Polyspace Projects	3-11
Troubleshoot Polyspace Analysis from MATLAB	3-13
Prerequisites	3-13
Capture Polyspace Analysis Errors in Error Log	3-13

Offload Polyspace Analysis to Remote Servers from Desktop

4

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

Run Polyspace Analysis on Generated Code

5

Run Polyspace Analysis on Code Generated with Embedded Coder	5-2
Prerequisites	5-2
Generate and Analyze Code	5-2
Review Analysis Results	5-4
Changes in Polyspace Analysis Workflows in Simulink in R2019b	5-7
Code Verification Workflow in a Nutshell	5-7
Locate Pre-R2019b Menu Items in Simulink Toolstrip	5-8
Run Polyspace Analysis on Code Generated from Simulink Model	5-10
Prerequisites	5-10
Open Model for Code Generation and Polyspace Analysis	5-10
Generate Code	5-11
Analyze Code	5-11
Review Analysis Results	5-12
Trace Errors Back to Model and Fix Them	5-12
Check for Coding Rule Violations	5-14
Annotate Blocks to Justify Results	5-14
Fix Model Design Issues Found as Bugs in Generated Code	5-16
Prerequisites	5-16
Open Model with Stateflow Chart Containing Design Issues	5-16
Generate and Analyze Code	5-16
Fix Issues	5-17
Run Polyspace Analysis on S-Function Code	5-21
Prerequisites	5-21
S-Function Analysis Workflow	5-21
Compile S-Functions to Be Compatible with Polyspace	5-21
Example S-Function Analysis	5-21

Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow	
Charts	5-23
Prerequisites	5-23
C/C++ Function Called Once in Model	5-23
C/C++ Function Called Multiple Times in Model	5-27
Run Polyspace Analysis on Custom Code in C Function Block	5-31
Prerequisites	5-31
Open Model for Running Polyspace Analysis on Custom Code in C Function Block	5-31
Run Polyspace Analysis	5-32
Identify Issues in C Code	5-33
Fix Identified Issues	5-35
Recommended Model Configuration Parameters for Polyspace Analysis	5-37
Configure Advanced Polyspace Options in Simulink	5-39
Configure Options	5-39
Share and Reuse Configuration	5-41
How Polyspace Analysis of Generated Code Works	5-43
Default Polyspace Options for Code Generated with Embedded Coder ..	5-44
Default Options	5-44
Constraint Specification	5-44
Recommended Polyspace options for Verifying Generated Code	5-45
Hardware Mapping Between Simulink and Polyspace	5-45
External Constraints on Polyspace Analysis of Generated Code	5-46
Extract External Constraints from Model	5-46
Storage Classes Supported for Constraint Extraction	5-47
Run Polyspace Analysis on Code Generated with TargetLink	5-49
Configure and Run Analysis	5-49
Review Analysis Results	5-50
Default Polyspace Options for Code Generated with TargetLink	5-52
TargetLink Support	5-52
Default Options	5-52
Lookup Tables	5-52
Data Range Specification	5-53
Code Generation Options	5-53
Troubleshoot Navigation from Code to Model	5-54
Links from Code to Model Do Not Appear	5-54
Links from Code to Model Do Not Work	5-54
Your Model Already Uses Highlighting	5-55
Run Polyspace on C/C++ Code Generated from MATLAB Code	5-56
Prerequisites	5-56
Run Polyspace Analysis	5-56
Review Analysis Results	5-58
Run Analysis for Specific Design Range	5-59

Configure Advanced Polyspace Options in MATLAB Coder App	5-62
Configure Options	5-62
Share and Reuse Configuration	5-63

Run Polyspace Analysis in IDE Plugins

6

Run Polyspace Analysis in Eclipse	6-2
Configure and Run Analysis	6-3
Review Analysis Results	6-5
Specify Polyspace Compiler Options Through Eclipse Project	6-7
Eclipse Refers Directly to Your Compilation Toolchain	6-7
Eclipse Uses Your Compilation Toolchain Through Build Command	6-8

Running Polyspace on AUTOSAR Code

7

Using Polyspace in AUTOSAR Software Development	7-2
Check if Implementation of Software Components Follow Specifications	7-2
Assess Impact of Edits to Specifications	7-3
Check Code Implementation for Run-time Errors and Mismatch with Specifications	7-3
Check Code Implementation Against Specification Updates	7-4
Benefits of Polyspace for AUTOSAR	7-5
Polyspace Modularizes Analysis Based on AUTOSAR Components	7-5
Polyspace Detects Mismatch Between Code and AUTOSAR XML Spec ...	7-8
Run Polyspace on AUTOSAR Code	7-12
Run Polyspace in User Interface	7-12
Run Polyspace Using Scripts	7-15
Troubleshoot Polyspace Analysis of AUTOSAR Code	7-17
View Project Completion Status	7-17
View Errors in AUTOSAR XML Parsing	7-18
View Compilation Errors in Code	7-18
Run Polyspace on AUTOSAR Code with Conservative Assumptions	7-21
Run Polyspace on AUTOSAR Code Using Build Command	7-23
Run Code Prover Without Compilation Options	7-23
Run Code Prover with Compilation Options from Build Command	7-24

8

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

Configure Target and Compiler Options

9

Specify Target Environment and Compiler Behavior	9-2
Extract Options from Build Command	9-2
Specify Options Explicitly	9-3
C/C++ Language Standard Used in Polyspace Analysis	9-5
Supported Language Standards	9-5
Default Language Standard	9-5
C11 Language Elements Supported in Polyspace	9-7
C++11 Language Elements Supported in Polyspace	9-8
C++14 Language Elements Supported in Polyspace	9-11
Provide Standard Library Headers for Polyspace Analysis	9-14
Requirements for Project Creation from Build Systems	9-15
Compiler Requirements	9-15
Build Command Requirements	9-16
Emulate Microchip MPLAB XC16 and XC32 Compilers	9-18
Supported Keil or IAR Language Extensions	9-19
Special Function Register Data Type	9-19
Keywords Removed During Preprocessing	9-20
Remove or Replace Keywords Before Compilation	9-21
Remove Unrecognized Keywords	9-21
Remove Unrecognized Function Attributes	9-23
Gather Compilation Options Efficiently	9-24

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

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

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

Configure Coding Rules Checking and Code Metrics Computation

12

Check for Coding Standard Violations	12-2
Configure Coding Rules Checking	12-2
Review Coding Rule Violations	12-6
Generate Reports	12-7
Avoid Violations of MISRA C 2012 Rules 8.x	12-8
Software Quality Objective Subsets (C:2004)	12-11
Rules in SQO-Subset1	12-11
Rules in SQO-Subset2	12-12
Software Quality Objective Subsets (AC AGC)	12-15
Rules in SQO-Subset1	12-15
Rules in SQO-Subset2	12-15
Software Quality Objective Subsets (C:2012)	12-18
Guidelines in SQO-Subset1	12-18
Guidelines in SQO-Subset2	12-19
Software Quality Objective Subsets (C++)	12-21
SQO Subset 1 - Direct Impact on Selectivity	12-21
SQO Subset 2 - Indirect Impact on Selectivity	12-22
Coding Rule Subsets Checked Early in Analysis	12-27
MISRA C: 2004 and MISRA AC AGC Rules	12-27
MISRA C: 2012 Rules	12-34
Create Custom Coding Rules	12-42
User Interface (Desktop Products Only)	12-42
Command Line	12-43
Compute Code Complexity Metrics	12-44
Impose Limits on Metrics (Desktop Products Only)	12-44
Impose Limits on Metrics (Server and Access products)	12-46

HIS Code Complexity Metrics	12-47
Project	12-47
File	12-47
Function	12-47

Coding Rule Sets and Concepts

13

Polyspace MISRA C 2004 and MISRA AC AGC Checkers	13-2
MISRA C:2004 and MISRA AC AGC Coding Rules	13-3
Supported MISRA C:2004 and MISRA AC AGC Rules	13-3
Troubleshooting	13-3
List of Supported Coding Rules	13-3
Unsupported MISRA C:2004 and MISRA AC AGC Rules	13-32
Polyspace MISRA C:2012 Checkers	13-34
Essential Types in MISRA C: 2012 Rules 10.x	13-35
Categories of Essential Types	13-35
How MISRA C: 2012 Uses Essential Types	13-35
Unsupported MISRA C:2012 Guidelines	13-37
Polyspace MISRA C++ Checkers	13-38
Unsupported MISRA C++ Coding Rules	13-39
Language Independent Issues	13-39
General	13-40
Lexical Conventions	13-40
Expressions	13-40
Declarations	13-41
Classes	13-41
Templates	13-41
Exception Handling	13-41
Library Introduction	13-42
Polyspace JSF C++ Checkers	13-43
JSF C++ Coding Rules	13-44
Supported JSF C++ Coding Rules	13-44
Unsupported JSF++ Rules	13-59

Configure Verification of Modules or Libraries

14

Provide Context for C Code Verification	14-2
Control Variable Range	14-2
Control Function Call Sequence	14-2

Control Stubbing Behavior	14-2
Provide Context for C++ Code Verification	14-4
Control Variable Range	14-4
Control Function Call Sequence	14-4
Verify C Application Without main Function	14-6
Generate main Function	14-6
Manually Write main Function	14-6
Verify C++ Classes	14-9
Verification of Classes	14-9
Methods and Class Specifics	14-11

Configure Comment Import from Previous Results

15

Import Review Information from Previous Polyspace Analysis	15-2
Automatic Import from Last Analysis	15-2
Import from Another Analysis Result	15-2
Import Algorithm	15-3
View Imported Review Information That Does Not Apply	15-3
Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results	15-5
Mapping Multiple MISRA C: 2004 Annotations to the Same MISRA C: 2012 Result	15-6

Interpret Polyspace Code Prover Results

16

Interpret Polyspace Code Prover Results	16-2
Interpret Result	16-3
Find Root Cause of Result	16-4
Code Prover Result and Source Code Colors	16-8
Result Colors	16-8
Source Code Colors	16-10
Global Variable Colors	16-11
Code Prover Run-Time Checks	16-13
Data Flow Checks	16-13
Numerical Checks	16-13
Static Memory Checks	16-14
Control Flow Checks	16-14
C++ Checks	16-14
Other Checks	16-15
Dashboard	16-16

Concurrency Modeling	16-20
Results List	16-21
Source	16-24
Result Details	16-30
Call Hierarchy	16-33
Variable Access	16-36
Code Prover Analysis Following Red and Orange Checks	16-42
Code Following Red Check	16-42
Green Check Following Orange Check	16-43
Gray Check Following Orange Check	16-43
Red Check Following Orange Check	16-44
Red Checks in Unreachable Code	16-44
Order of Code Prover Run-Time Checks	16-46
Orange Checks in Code Prover	16-48
When Orange Checks Occur	16-48
Why Review Orange Checks	16-49
How to Review Orange Checks	16-49
How to Reduce Orange Checks	16-49
Managing Orange Checks	16-51
Software Development Stage	16-52
Quality Goals	16-53
Critical Orange Checks	16-55
Path	16-55
Bounded Input Values	16-56
Unbounded Input Values	16-56
Limit Display of Orange Checks	16-57
Software Quality Objectives	16-60
Comparing Verification Results Against Software Quality Objectives	16-65
Reduce Orange Checks	16-67
Provide Context for Verification	16-67
Improve Verification Precision	16-68
Follow Coding Rules	16-68
Reduce Application Size	16-69
Test Orange Checks for Run-Time Errors	16-70
Run Tests for Full Range of Input	16-70
Run Tests for Specified Range of Input	16-71
Limitations of Automatic Orange Tester	16-73
Unsupported Platforms	16-73
Unsupported Polyspace Options	16-73
Options with Restrictions	16-73

Reviewing Checks

17

Review and Fix Absolute Address Usage Checks	17-2
Review and Fix Correctness Condition Checks	17-3
Step 1: Interpret Check Information	17-3
Step 2: Determine Root Cause of Check	17-5
Step 3: Trace Check to Polyspace Assumption	17-6
Review and Fix Division by Zero Checks	17-7
Step 1: Interpret Check Information	17-7
Step 2: Determine Root Cause of Check	17-8
Step 3: Look for Common Causes of Check	17-9
Step 4: Trace Check to Polyspace Assumption	17-10
Review and Fix Function Not Called Checks	17-11
Step 1: Interpret Check Information	17-11
Step 2: Determine Root Cause of Check	17-11
Step 3: Look for Common Causes of Check	17-12
Review and Fix Function Not Reachable Checks	17-13
Step 1: Interpret Check Information	17-13
Step 2: Determine Root Cause of Check	17-13
Review and Fix Function Not Returning Value Checks	17-15
Step 1: Interpret Check Information	17-15
Step 2: Determine Root Cause of Check	17-15
Review and Fix Illegally Dereferenced Pointer Checks	17-16
Step 1: Interpret Check Information	17-16
Step 2: Determine Root Cause of Check	17-18
Step 3: Look for Common Causes of Check	17-20
Step 4: Trace Check to Polyspace Assumption	17-21
Review and Fix Incorrect Object Oriented Programming Checks	17-22
Step 1: Interpret Check Information	17-22
Step 2: Determine Root Cause of Check	17-22
Review and Fix Invalid C++ Specific Operations Checks	17-24
Step 1: Interpret Check Information	17-24
Step 2: Determine Root Cause of Check	17-24
Step 3: Trace Check to Polyspace Assumption	17-25
Review and Fix Invalid Shift Operations Checks	17-26
Step 1: Interpret Check Information	17-26
Step 2: Determine Root Cause of Check	17-27
Step 3: Look for Common Causes of Check	17-29
Step 4: Trace Check to Polyspace Assumption	17-29

Review and Fix Invalid Use of Standard Library Routine Checks	17-30
Step 1: Interpret Check Information	17-30
Step 2: Trace Check to Polyspace Assumption	17-31
Invalid Use of Standard Library Floating Point Routines	17-32
What the Check Looks For	17-32
Single-Argument Functions Checked	17-33
Functions with Multiple Arguments	17-33
Review and Fix Non-initialized Local Variable Checks	17-35
Step 1: Interpret Check Information	17-35
Step 2: Determine Root Cause of Check	17-35
Step 3: Look for Common Causes of Check	17-36
Step 4: Trace Check to Polyspace Assumption	17-37
Review and Fix Non-initialized Pointer Checks	17-38
Step 1: Interpret Check Information	17-38
Step 2: Determine Root Cause of Check	17-38
Step 3: Trace Check to Polyspace Assumption	17-39
Review and Fix Non-initialized Variable Checks	17-40
Step 1: Interpret Check Information	17-40
Step 2: Determine Root Cause of Check	17-40
Step 3: Trace Check to Polyspace Assumption	17-41
Review and Fix Non-Terminating Call Checks	17-42
Step 1: Determine Root Cause of Check	17-42
Step 2: Look for Common Causes of Check	17-42
Identify Function Call with Run-Time Error	17-44
Review and Fix Non-Terminating Loop Checks	17-46
Step 1: Interpret Check Information	17-46
Step 2: Determine Root Cause of Check	17-46
Step 3: Look for Common Causes of Check	17-47
Identify Loop Operation with Run-Time Error	17-49
Review and Fix Null This-pointer Calling Method Checks	17-51
Step 1: Interpret Check Information	17-51
Step 2: Determine Root Cause of Check	17-51
Review and Fix Out of Bounds Array Index Checks	17-53
Step 1: Interpret Check Information	17-53
Step 2: Determine Root Cause of Check	17-53
Step 3: Look for Common Causes of Check	17-55
Step 4: Trace Check to Polyspace Assumption	17-55
Review and Fix Overflow Checks	17-57
Step 1: Interpret Check Information	17-57
Step 2: Determine Root Cause of Check	17-57
Step 3: Look for Common Causes of Check	17-59
Step 4: Trace Check to Polyspace Assumption	17-60
Detect Overflows in Buffer Size Computation	17-61

Review and Fix Return Value Not Initialized Checks	17-63
Step 1: Interpret Check Information	17-63
Step 2: Determine Root Cause of Check	17-63
Step 3: Look for Common Causes of Check	17-64
Step 4: Trace Check to Polyspace Assumption	17-65
Review and Fix Uncaught Exception Checks	17-66
Step 1: Interpret Check Information	17-66
Step 2: Determine Root Cause of Check	17-66
Review and Fix Unreachable Code Checks	17-68
Step 1: Interpret Check Information	17-68
Step 2: Determine Root Cause of Check	17-68
Step 3: Look for Common Causes of Check	17-70
Review and Fix User Assertion Checks	17-72
Step 1: Determine Root Cause of Check	17-72
Step 2: Look for Common Causes of Check	17-74
Step 3: Trace Check to Polyspace Assumption	17-74
Find Relations Between Variables in Code	17-75
Insert Pragma to Determine Variable Relation	17-75
Further Exploration	17-77
Review Polyspace Results on AUTOSAR Code	17-78
Open Results	17-78
See Overview of Results for all Software Components	17-79
See Runnables and Source Files in Software Component	17-80
Interpret AUTOSAR Specific Run-time Checks for Software Component	17-83

Fix or Comment Polyspace Results

18

Address Polyspace Results Through Bug Fixes or Justifications	18-2
Add Review Information to Results File	18-2
Comment or Annotate in Code	18-3
Annotate Code and Hide Known or Acceptable Results	18-6
Code Annotation Syntax	18-6
Syntax Examples	18-9
Short Names of Code Prover Run-Time Checks	18-12
Short Names of Code Complexity Metrics	18-14
Project Metrics	18-14
File Metrics	18-14
Function Metrics	18-14
Annotate Code for Known or Acceptable Results (Not Recommended)	18-16
Add Annotations from the Polyspace Interface	18-16

Add Annotations Manually	18-17
Define Custom Annotation Format	18-20
Define Annotation Syntax Format	18-22
Map Your Annotation to the Polyspace Annotation Syntax	18-25
Annotation Description Full XML Template	18-27
Example	18-30
Justify Coding Rule Violations Using Code Prover Checks	18-33
Rules About Data Type Conversions	18-33
Rules About Pointer Arithmetic	18-35

Manage Results

19

Filter and Group Results	19-2
Filter Results	19-3
Group Results	19-7
Prioritize Check Review	19-9

Generate Reports from Polyspace Results

20

Generate Reports	20-2
Generate Reports from User Interface	20-2
Generate Reports from Command Line	20-3
Export Polyspace Analysis Results	20-5
Export Results to Text File	20-5
Export Results to MATLAB Table	20-6
View Exported Results	20-6
Export Polyspace Analysis Results to Excel by Using MATLAB Scripts	20-7
Report Result Summary and Details in One Worksheet	20-7
Control Formatting of Excel Report	20-8
Export Global Variable List	20-9
Export Variable List to Text File	20-9
Export Variable List to MATLAB Table	20-10
View Exported Variable List	20-10
Visualize Code Prover Analysis Results in MATLAB	20-13
Export Results to MATLAB Table	20-13
Generate Graphs from Results and Include in Report	20-13

Customize Existing Code Prover Report Template	20-16
Prerequisites	20-16
View Components of Template	20-16
Change Components of Template	20-17
Further Exploration	20-20
Sample Report Template Customizations	20-21
Add List of Recursive Functions	20-21
Show Red Run-Time Checks Only	20-21
Show Non-Justified Run-Time Checks Only	20-22
Add Chapter for Functional Design Errors	20-23

Software Quality with Polyspace Metrics

21

Code Quality Metrics	21-2
Summary Tab	21-2
Code Metrics Tab	21-4
Coding Rules Tab	21-4
Run-Time Checks Tab	21-5
Generate Code Quality Metrics	21-9
Upload Results to Polyspace Metrics After Remote Verification	21-9
Upload Local Verification Results to Polyspace Metrics	21-9
View Code Quality Metrics	21-12
Open Metrics Interface	21-12
View All Projects and Runs	21-12
Review Metrics for Particular Project or Run	21-13
Compare Metrics Against Software Quality Objectives	21-15
Apply Predefined Objectives to Metrics	21-15
Customize Software Quality Objectives	21-17
View Trends in Code Quality Metrics	21-20
Web Browser Requirements for Polyspace Metrics	21-23
Additional Considerations	21-23
Elements in Custom Software Quality Objectives File	21-24
HIS Metrics	21-24
Non-HIS Metrics	21-24

Troubleshooting in Polyspace Code Prover

22

View Error Information When Analysis Stops	22-3
View Error Information in User Interface	22-3
View Error Information in Log File	22-4

Troubleshoot Compilation and Linking Errors	22-6
Issue	22-6
Possible Cause: Deviations from Standard	22-6
Possible Cause: Linking Errors	22-7
Possible Cause: Conflicts with Polyspace Function Stubs	22-8
Reduce Memory Usage and Time Taken by Polyspace Analysis	22-10
Issue	22-10
Possible Cause: Temporary Folder on Network Drive	22-10
Possible Cause: Anti-Virus Software	22-10
Possible Cause: Large and Complex Application	22-11
Possible Cause: Too Many Entry Points for Multitasking Applications ..	22-13
Understand Verification Results	22-15
Issue	22-15
Possible Cause: Relation to Prior Code Operations	22-15
Possible Cause: Software Assumptions	22-16
Contact Technical Support About Issues with Running Polyspace	22-18
Provide System Information	22-18
Provide Information About the Issue	22-18
Polyspace Cannot Find the Server	22-21
Message	22-21
Possible Cause	22-21
Solution	22-21
Job Manager Cannot Write to Database	22-22
Message	22-22
Possible Cause	22-22
Workaround	22-22
Compiler Not Supported for Project Creation from Build Systems . . .	22-23
Issue	22-23
Cause	22-23
Solution	22-23
Slow Build Process When Polyspace Traces the Build	22-29
Issue	22-29
Cause	22-29
Solution	22-29
Check if Polyspace Supports Build Scripts	22-30
Issue	22-30
Possible Cause	22-30
Solution	22-30
Troubleshooting Project Creation from MinGW Build	22-32
Issue	22-32
Cause	22-32
Solution	22-32
Troubleshooting Project Creation from Visual Studio Build	22-33

Error Processing Macro with Semicolon in Build System	22-34
Issue	22-34
Cause	22-34
Solution	22-34
Could Not Find Include File	22-35
Issue	22-35
Cause	22-35
Solution	22-35
Conflicting Universal Unique Identifiers (UUIDs)	22-37
Issue	22-37
Solution	22-37
Data Type Not Recognized	22-38
Issue	22-38
Cause	22-38
Solution	22-38
Undefined Identifier Error	22-40
Issue	22-40
Possible Cause: Missing Files	22-40
Possible Cause: Unrecognized Keyword	22-40
Possible Cause: Declaration Embedded in #ifdef Statements	22-41
Possible Cause: Project Created from Non-Debug Build	22-41
Unknown Function Prototype Error	22-43
Issue	22-43
Cause	22-43
Solution	22-43
Error Related to #error Directive	22-44
Issue	22-44
Cause	22-44
Solution	22-44
Large Object Error	22-45
Issue	22-45
Cause	22-45
Solution	22-45
Errors Related to Generic Compiler	22-47
Issue	22-47
Cause	22-47
Solution	22-47
Errors Related to Keil or IAR Compiler	22-48
Missing Identifiers	22-48
Errors Related to Diab Compiler	22-49
Issue	22-49
Cause	22-49
Solution	22-49

Errors Related to Green Hills Compiler	22-51
Issue	22-51
Cause	22-51
Solution	22-51
Errors Related to TASKING Compiler	22-53
Issue	22-53
Cause	22-53
Solution	22-53
Errors from In-Class Initialization (C++)	22-55
Errors from Double Declarations of Standard Template Library Functions (C++)	22-56
Errors Related to GNU Compiler	22-57
Issue	22-57
Cause	22-57
Solution	22-57
Errors Related to Visual Compilers	22-58
Import Folder	22-58
pragma Pack	22-58
C++/CLI	22-59
Conflicting Declarations in Different Translation Units	22-60
Issue	22-60
Possible Cause: Variable Declaration and Definition Mismatch	22-61
Possible Cause: Function Declaration and Definition Mismatch	22-61
Possible Cause: Conflicts from Unrelated Declarations	22-62
Possible Cause: Macro-dependent Definitions	22-63
Possible Cause: Keyword Redefined as Macro	22-63
Possible Cause: Differences in Structure Packing	22-64
Errors from Conflicts with Polyspace Header Files	22-65
Issue	22-65
Cause	22-65
Solution	22-65
C++ Standard Template Library Stubbing Errors	22-66
Issue	22-66
Cause	22-66
Solution	22-66
Lib C Stubbing Errors	22-67
Extern C Functions	22-67
Functional Limitations on Some Stubbed Standard ANSI Functions	22-67
Errors from Using Namespace std Without Prefix	22-69
Issue	22-69
Cause	22-69
Solution	22-69
Errors from Assertion or Memory Allocation Functions	22-70
Issue	22-70

Cause	22-70
Solution	22-70
Eclipse Java Version Incompatible with Polyspace Plug-in	22-71
Issue	22-71
Cause	22-71
Solution	22-71
Reasons for Unchecked Code	22-72
Issue	22-72
Possible Cause: Compilation Errors	22-73
Possible Cause: Early Red or Gray Check	22-73
Possible Cause: Incorrect Options	22-75
Possible Cause: main Function Does Not Terminate	22-75
Source Files or Functions Not Displayed in Results List	22-78
Issue	22-78
Possible Cause: Files Not Verified	22-78
Possible Cause: Filters Applied	22-79
Coding Standard Violations Not Displayed	22-81
Issue	22-81
Possible Cause: Rule Checker Not Enabled	22-81
Possible Cause: Rule Violations in Header Files	22-81
Possible Cause: Rule Violations in Macros	22-81
Possible Cause: Rule Violations in Sources Because of Header Files ...	22-82
Possible Cause: Compilation Errors	22-82
Possible Cause: Code Prover Analysis with Lower Verification Level ...	22-82
Incorrect Behavior of Standard Library Math Functions	22-83
Issue	22-83
Cause	22-83
Solution	22-83
Insufficient Memory During Report Generation	22-84
Message	22-84
Possible Cause	22-84
Solution	22-84
Errors with Temporary Files	22-85
No Access Rights	22-85
No Space Left on Device	22-85
Cannot Open Temporary File	22-85
Error or Slow Runs from Disk Defragmentation and Anti-virus Software	
.....	22-87
Issue	22-87
Possible Cause	22-87
Solution	22-87
SQLite I/O Error	22-89
Issue	22-89
Cause	22-89
Solution	22-89

License Error -4,0	22-90
Issue	22-90
Possible Cause: Another Polyspace Instance Running	22-90
Possible Cause: Prior Polyspace Run in Simulink or MATLAB Coder . . .	22-90

Run Polyspace Analysis on Desktop

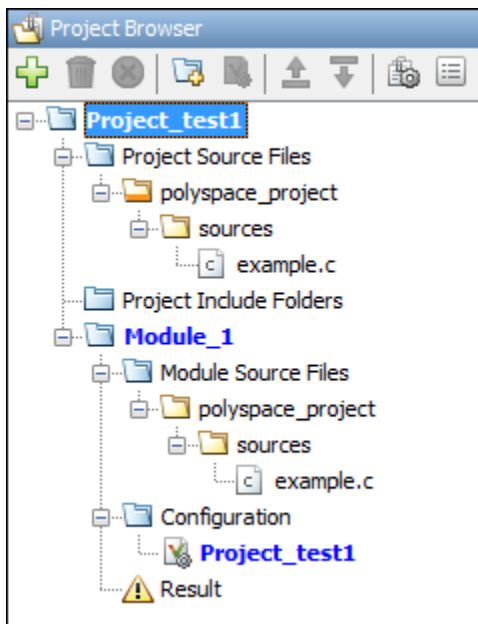
- “Add Source Files for Analysis in Polyspace User Interface” on page 1-2
- “Run Polyspace Analysis on Desktop” on page 1-7
- “Project and Results Folder Contents” on page 1-11
- “Storage of Temporary Files” on page 1-12
- “Create Project Using Visual Studio Information” on page 1-13
- “Create Project Using Configuration Template” on page 1-15
- “Update Polyspace Project” on page 1-18
- “Organize Layout of Polyspace User Interface” on page 1-21
- “Customize Polyspace User Interface” on page 1-23
- “Upload Results to Polyspace Access” on page 1-27

Add Source Files for Analysis in Polyspace User Interface

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

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

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



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

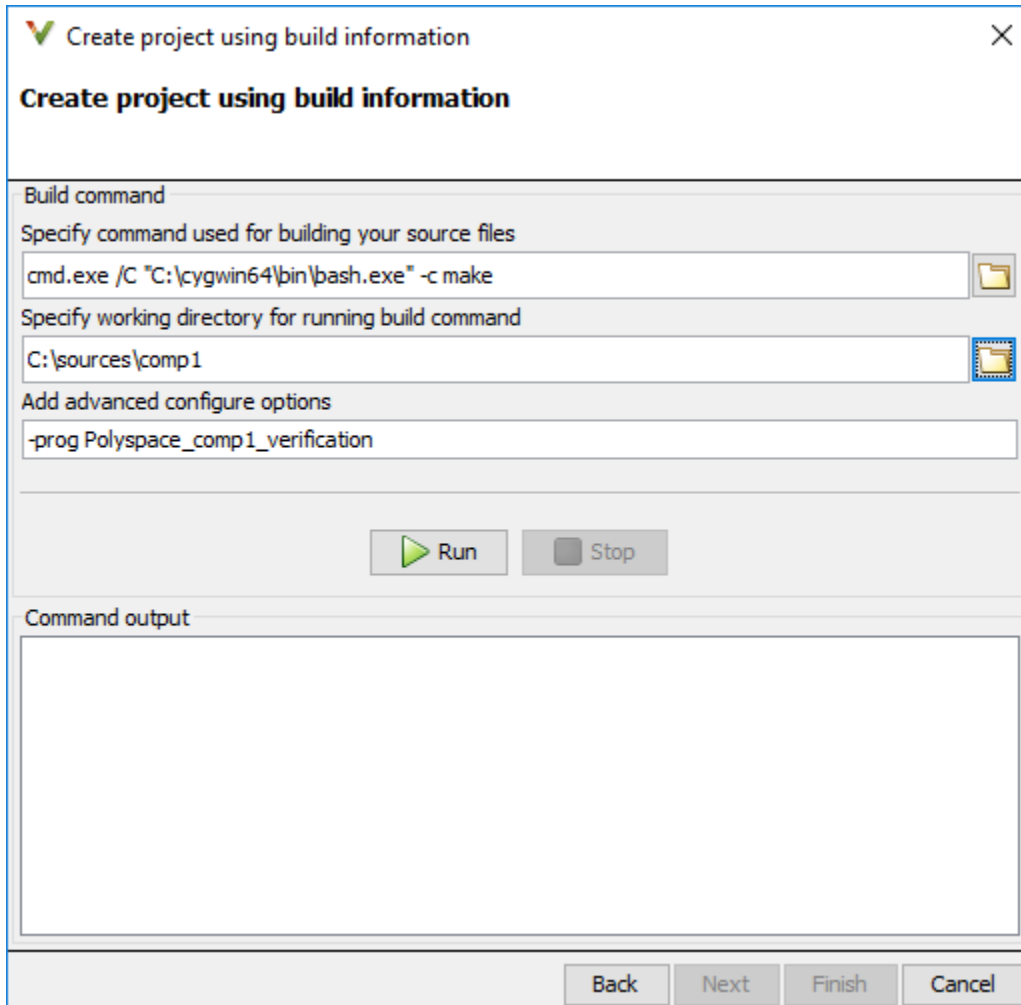
Note that when you reopen a project, the source file paths are computed based on the project location. For instance, suppose you commit the Polyspace project to a version control system along with your source files. When you check out the project from your version control system and open a local copy of the project, all source file paths are recomputed based on the new location of the project. The project now points to a local copy of the source files.

Add Sources from Build Command

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

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

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



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

If you build your source code within an IDE such as Visual Studio®, in the field for specifying the build command, enter the path to your executable, for instance, `C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\VCExpress.exe`. When you click **Run**, Polyspace opens your IDE. In your IDE, perform a complete build of your code. When you close your IDE, Polyspace extracts your source paths and compiler information. See also "Create Project Using Visual Studio Information" on page 1-13.

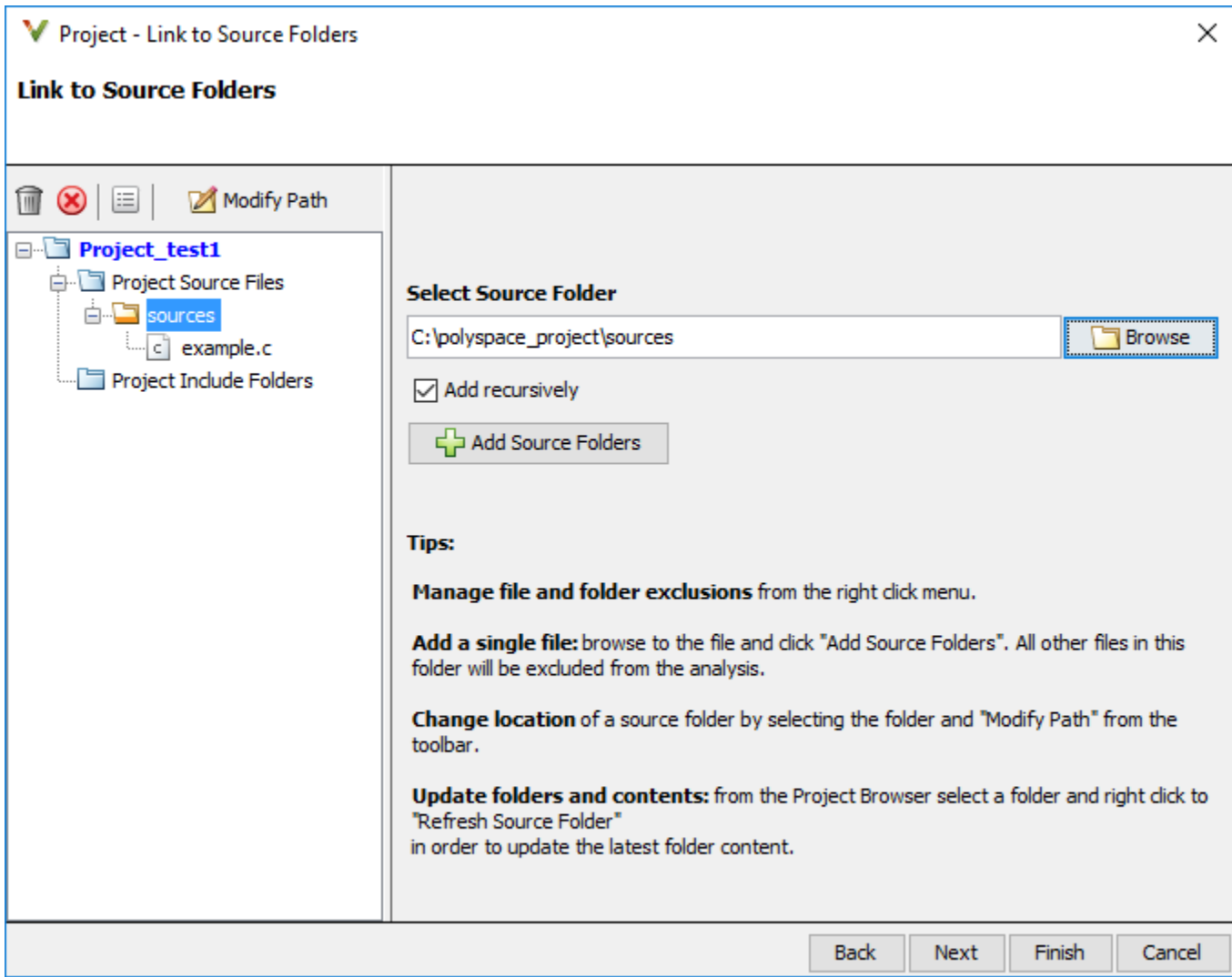
When you create a project from your build command, the **Project Browser** pane displays your source folders but not the include folders. In case you want to verify that your include folders were extracted, open the project file (with extension `.psprj`) in a text editor.

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

Add Sources Manually

Select **File > New Project**.

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



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

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

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

For Standard Library headers such as `stdio.h`, if you know the path to the headers from your compiler, specify them explicitly. Otherwise, the analysis uses Polyspace implementation of the Standard Library headers, which in some special cases, might not match your compiler implementation. See also “Provide Standard Library Headers for Polyspace Analysis” on page 9-14.

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

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

See Also

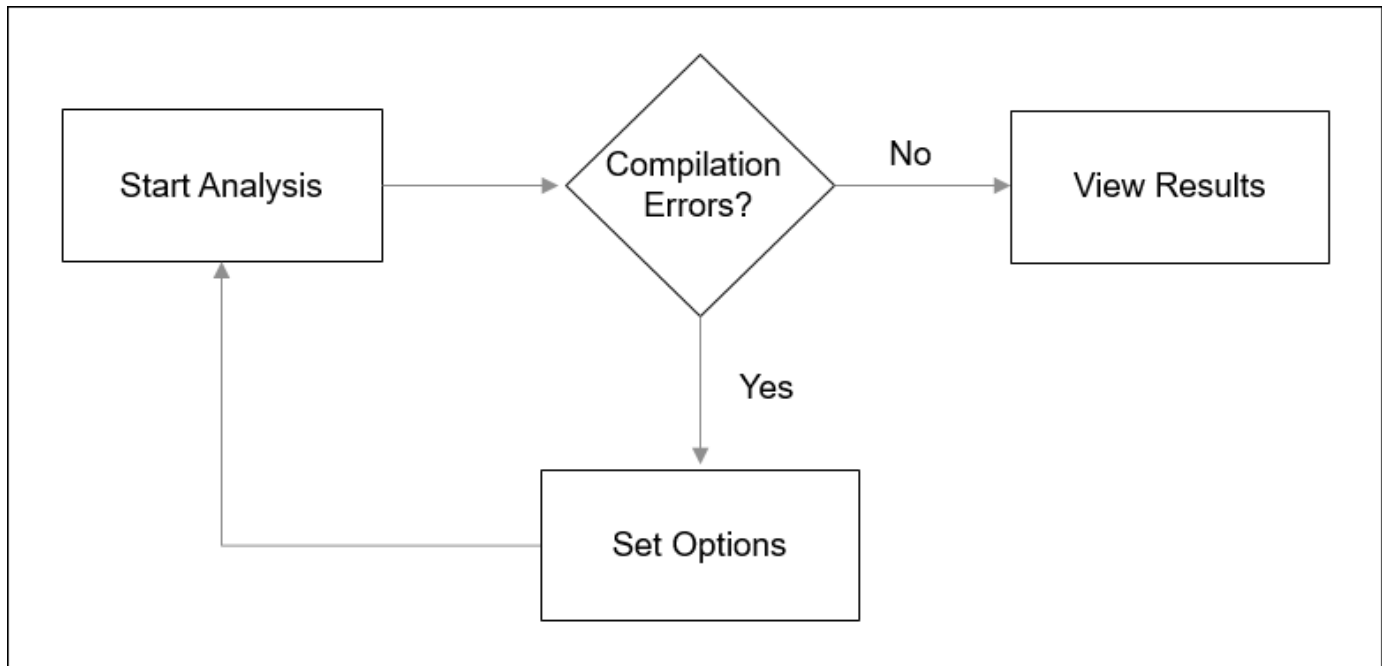
More About

- “Run Polyspace Analysis on Desktop” on page 1-7
- “Create Project Using Visual Studio Information” on page 1-13
- “Provide Standard Library Headers for Polyspace Analysis” on page 9-14

Run Polyspace Analysis on Desktop

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

After you specify your source files and compiler on page 1-2, start the Polyspace analysis. During analysis, Polyspace first compiles your code, and then checks for bugs (Bug Finder) or proves code correctness (Code Prover). If you encounter compilation errors, read the error message and diagnose the root cause of the error. To resolve the errors, you often have to set some Polyspace configuration options and rerun the analysis.



Arrange Layout of Windows for Project Setup

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

1 Run Polyspace Analysis on Desktop

➤ Select product.
➤ Start/stop analysis.

Set options as needed:

- Target & Compiler
- Macros
- Environment Settings

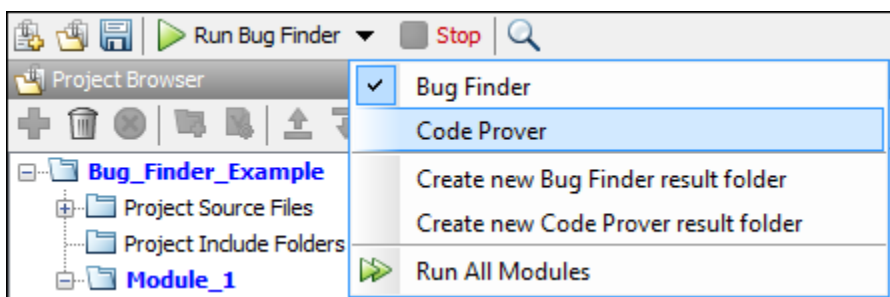
➤ Monitor progress
➤ Check for warnings and errors.

Select error message for more details.

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

Set Product and Result Location

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



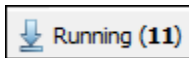
The results are stored in subfolders `Module_1`, `Module_2`, and so on in the project folder. To find the physical location of the project folder, right-click a project on the **Project Browser** pane and select **Open Folder with File Manager**.

To use a different folder naming convention or a different storage location for results, select **Tools > Preferences** and use the options on the **Project and Results Folder** tab.

Start and Monitor Analysis

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


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



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

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

Fix Compilation Errors

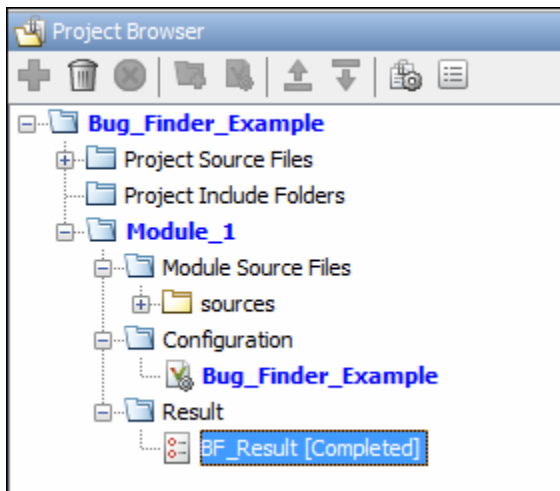
If compilation errors occur, the analysis continues on the remaining files that do compile. The **Dashboard** pane shows that some files did not compile and links to the **Output Summary** pane for details. The **Output Summary** pane shows compilation errors with a  icon.

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

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

Open Results

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



The Bug Finder (Code Prover) results are stored in a .psbf (.pscp) file in the results folder. For instance, if you save your project in C:\Projects\, a .psbf file for the Bug Finder analysis results on the first module Module_1 is stored in C:\Projects\Module_1\BF_Result.t. See also “Project and Results Folder Contents” on page 1-11.

See Also

More About

- “Run Polyspace Analysis from Command Line” on page 2-2
- “Run Polyspace Analysis by Using MATLAB Scripts” on page 3-5
- “Interpret Polyspace Code Prover Results” on page 16-2
- “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2
- “Filter and Group Results” on page 19-2

Project and Results Folder Contents

When you run an analysis in the Polyspace user interface, Polyspace generates files that contain information about configuration options and analysis results.

The organization of Polyspace files in the physical folder location follows the hierarchy displayed in the Polyspace user interface. 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:

- `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`.
- `Polyspace-Instrumented` — When the software runs the Automatic Orange Tester (AOT) at the end of a static verification, the software creates the `Polyspace-Instrumented` folder. The `Polyspace-Instrumented` folder contains files associated with the configuration and running of the Automatic Orange Tester.

See Also

`-results-dir`

Storage of Temporary Files

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

- “Errors with Temporary Files” on page 22-85
- “Reduce Memory Usage and Time Taken by Polyspace Analysis” on page 22-10

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`

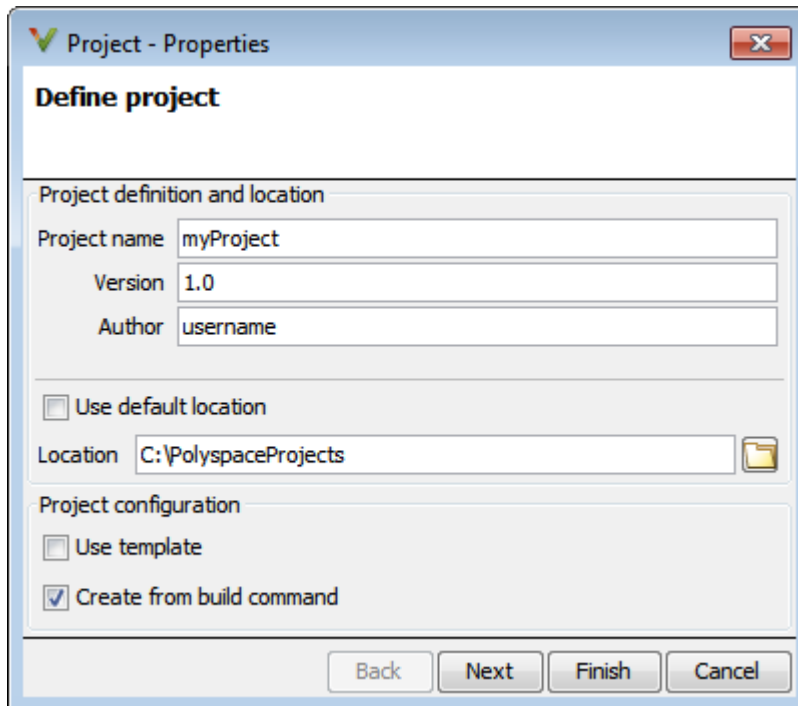
If one of these variables is defined, Polyspace uses that path for storing temporary files. If these environment variables are not defined, Polyspace stores temporary files in:


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

Create Project Using Visual Studio Information

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

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

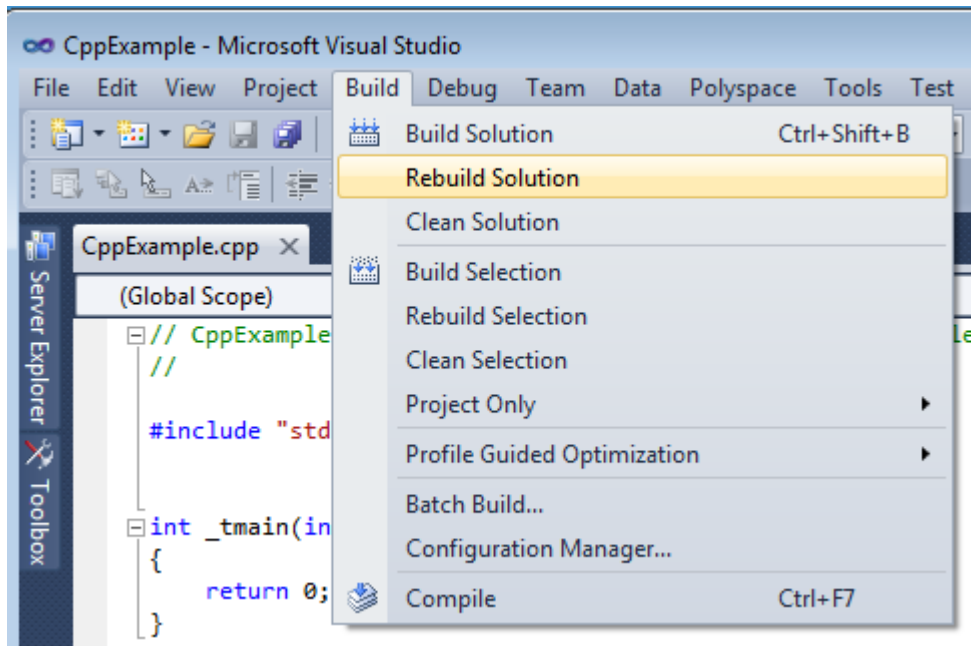


- 3 In the field **Specify command used for building your source files**, enter the full path to the Visual Studio executable. For instance, "C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\devenv.exe".
- 4 In the field **Specify working directory for running build command**, enter a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

This action opens the Visual Studio environment.

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

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



- 6 After the project builds, close Visual Studio.

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

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

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

See Also

More About

- “Troubleshooting Project Creation from Visual Studio Build” on page 22-33

Create Project Using Configuration Template

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

Why Use Templates

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

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

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

Polyspace software provides predefined templates for common compilers such as IAR, Kiel, Visual and VxWorks. For additional templates, see Polyspace Compiler Templates.

- Set analysis options manually. You can then save your options as a template and reuse them later. You can also share the template with other users and enforce consistent usage of Polyspace Bug Finder™ in your organization.

Use Predefined Template

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

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

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

Create Your Own Template

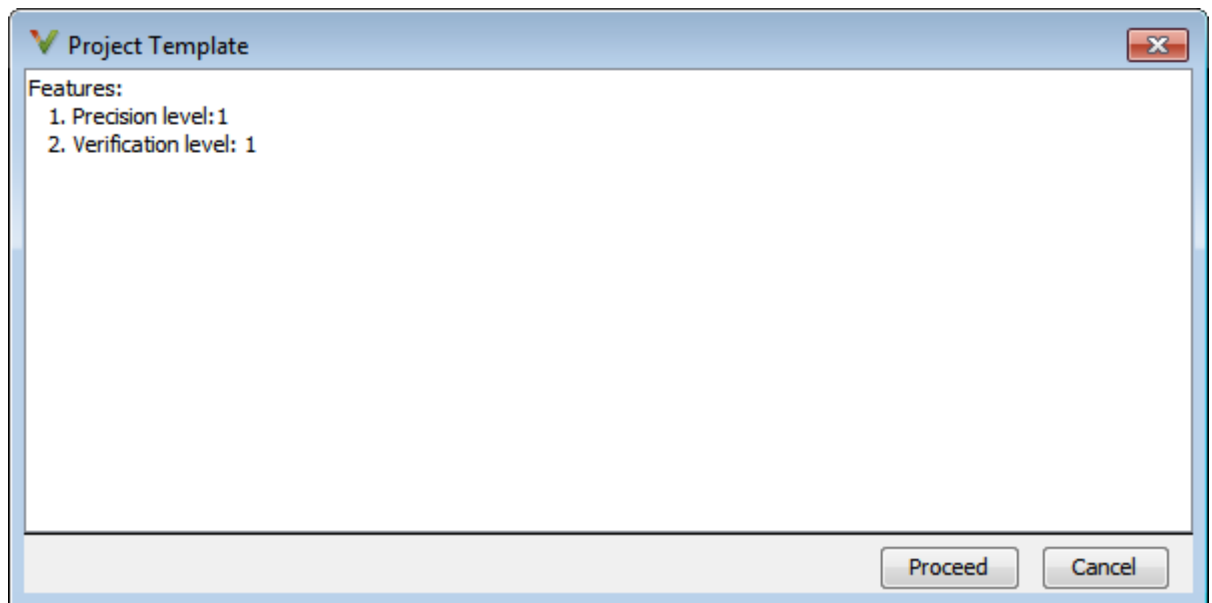
This example shows how to save a configuration from an existing project and create a new project using the saved configuration.


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

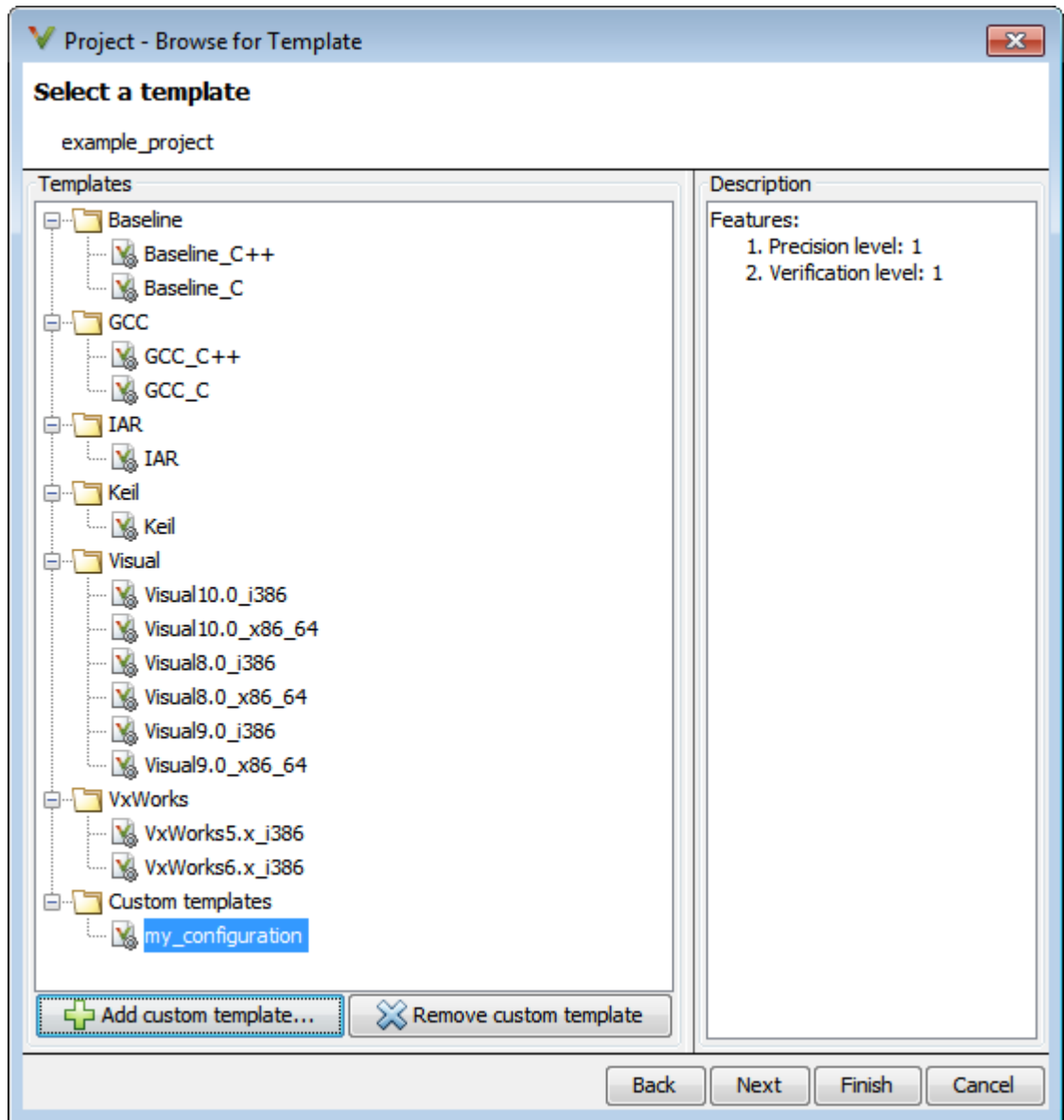
Suppose you create a Code Prover configuration template that runs Code Prover analysis to a precision level of 1 and a verification level of 1. See:

- Precision level (-0)
- Verification level (-to)

You can enter this description for the template.



- When you create a new project, to use a saved template:
 - 1** Select .
 - 2** Navigate to the template that you saved earlier, and then click **Open**. The new template appears in the **Custom templates** folder on the **Templates** browser. Select the template for use.



See Also

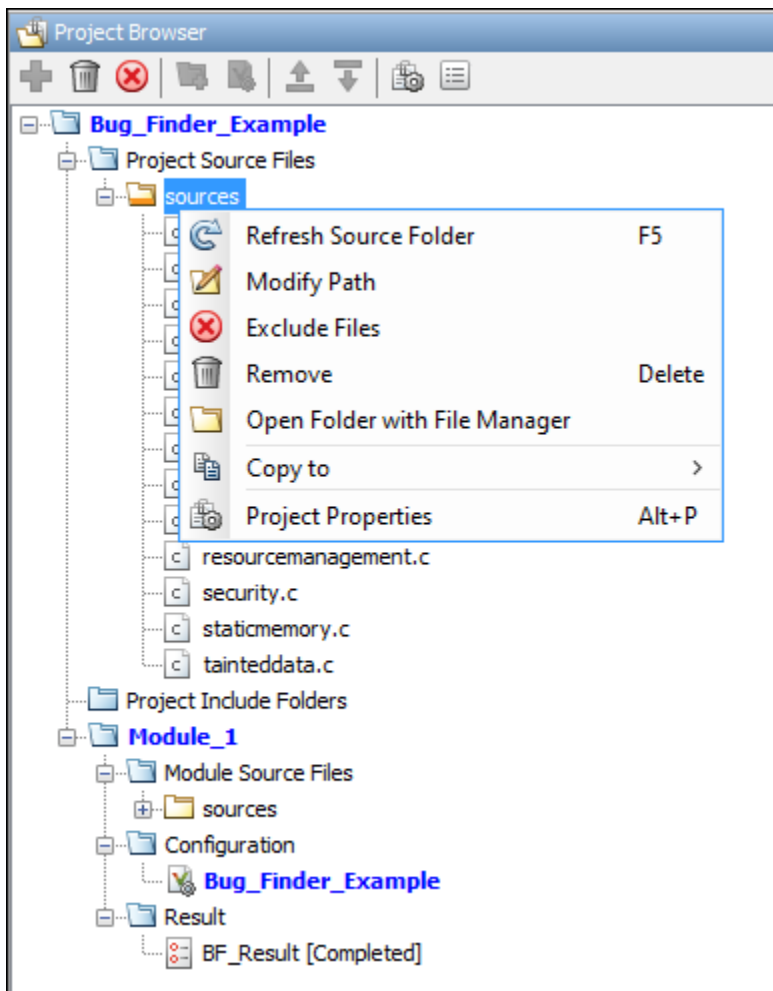
More About

- "Specify Polyspace Analysis Options" on page 8-2
- "Analysis Options"

Update Polyspace Project


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

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



Change Folder Path

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

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

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

Refresh Source List

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


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

Refresh Project Created from Build Command

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

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

Add Source and Include Folders

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

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

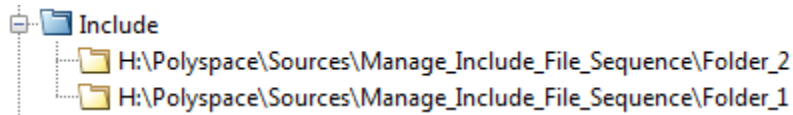
Before running an analysis, you must copy the source files to a module. Select the source files that you want to copy. To select multiple files together, press the **Ctrl** key while selecting the files. Right-click your selection. Select **Copy to > Module_n**. *n* is the module number.



Manage Include File Sequence

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

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

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



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

See Also

Related Examples

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

Organize Layout of Polyspace User Interface

The Polyspace user interface has two default layouts of panes.

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

Project Browser	Configuration
	Output Summary

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

Results List	Result Details
	Dashboard

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

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

Create Your Own Layout

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

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

To move a pane to another location:


1 Float the pane in one of three ways:

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


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

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

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

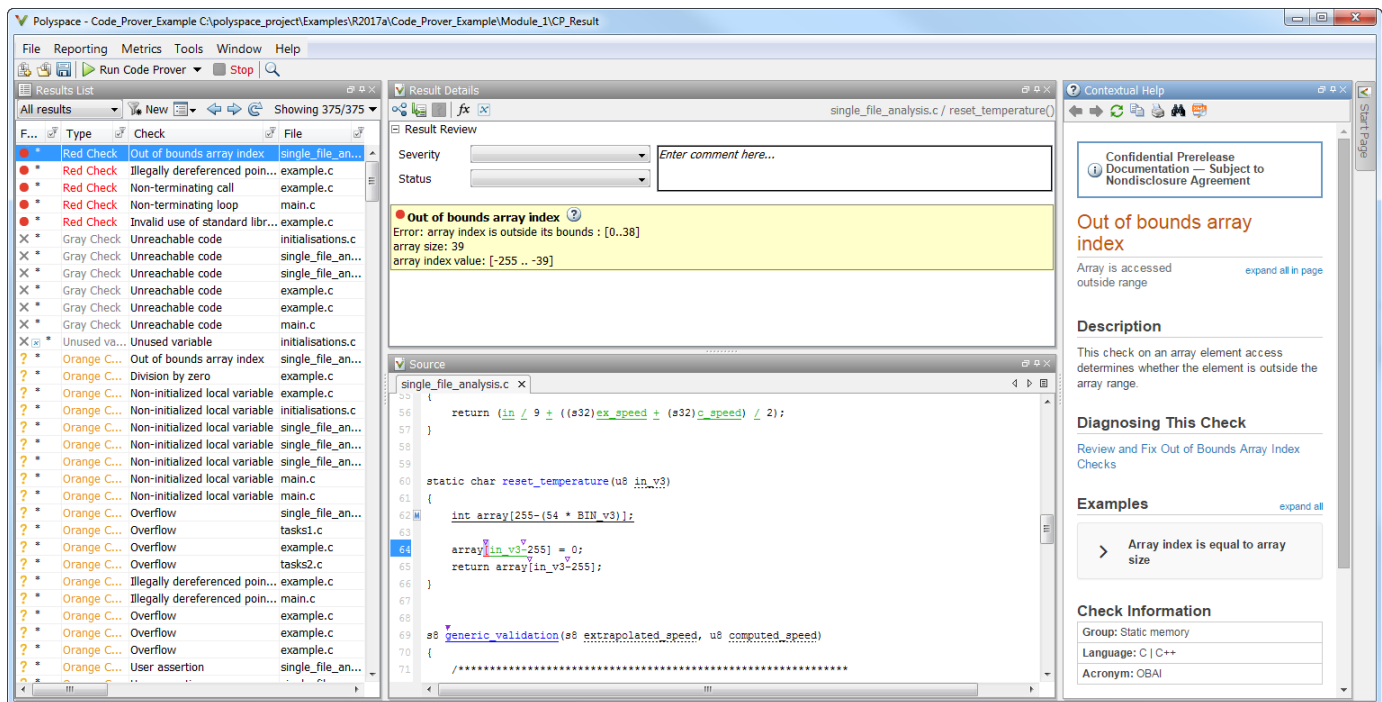
- Click  on the top right of the pane to float all tabs in that pane.

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

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

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

1 Run Polyspace Analysis on Desktop



Save and Reset Layout

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

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

See Also

More About

- “Customize Polyspace User Interface” on page 1-23
- “Organize Layout of Polyspace User Interface” on page 1-21

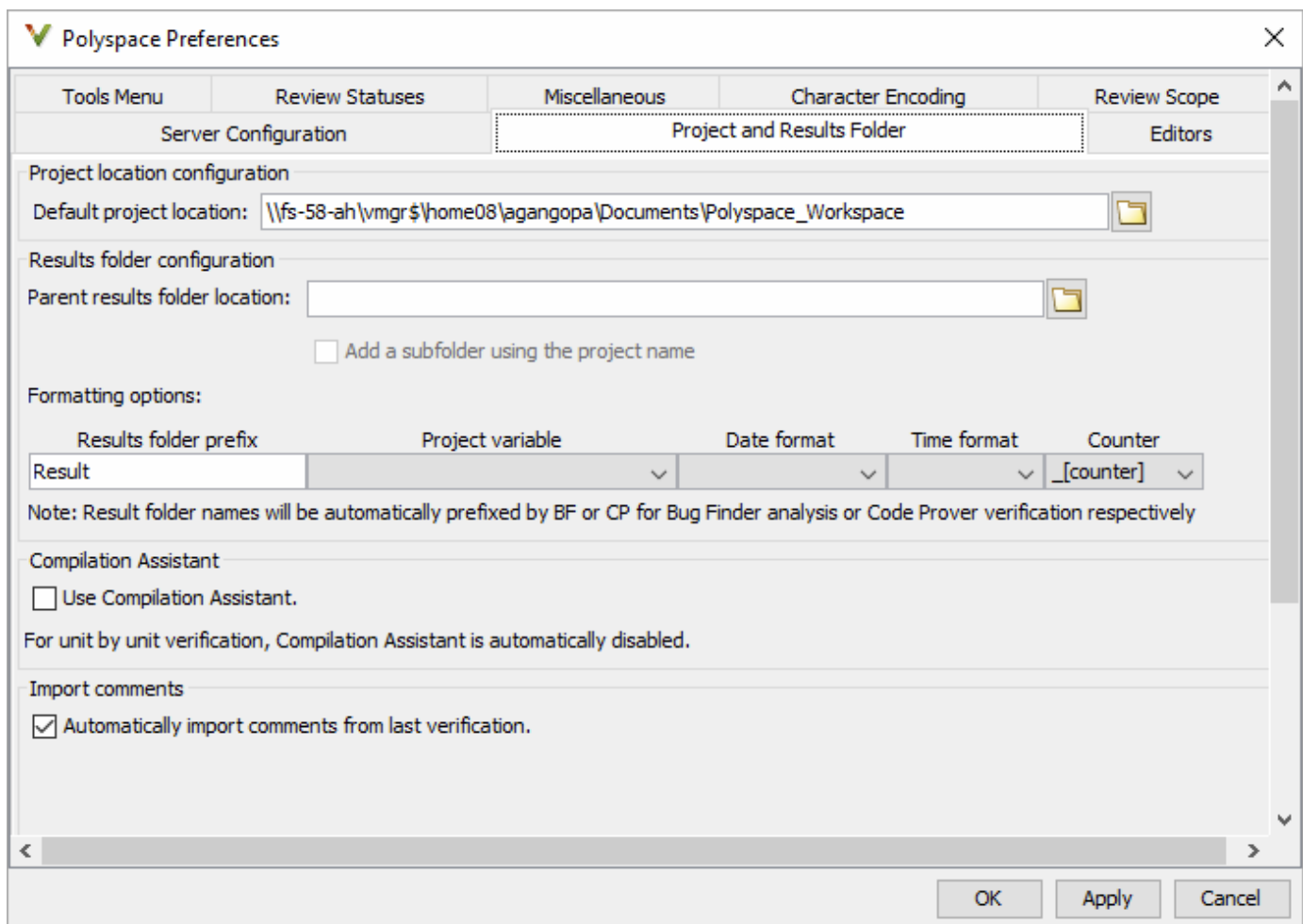
Customize Polyspace User Interface

In this section...

“Possible Customizations” on page 1-23

“Storage of Polyspace User Interface Customizations” on page 1-25

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



Possible Customizations

Change Default Font Size

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

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

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

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

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

Specify External Text Editor

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

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

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

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

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

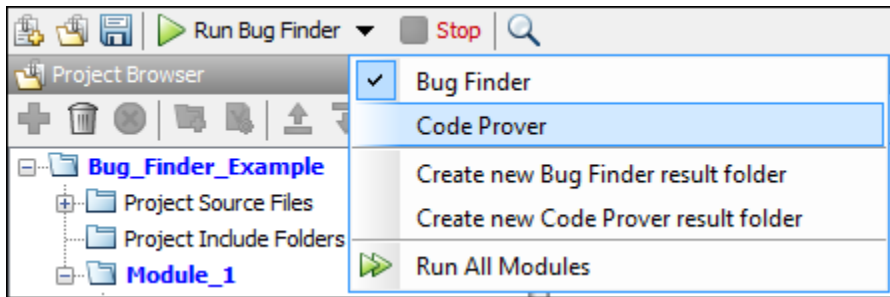
For console-based text editors, you must create a terminal. For example, to specify vi:

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

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

Create Naming Convention for Results Folder

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



If you create a new results folder for each run, you can define a naming convention for the folder. To specify a results folder naming convention, select the **Project and Results Folder** tab. In the section **Results folder configuration**, use the options under **Formatting options** to create a naming convention for results folders.

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

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

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

Create Custom Review Status

When reviewing Polyspace results, you can assign a status such as To fix or Justified. See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

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

Storage of Polyspace User Interface Customizations

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

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

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

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

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

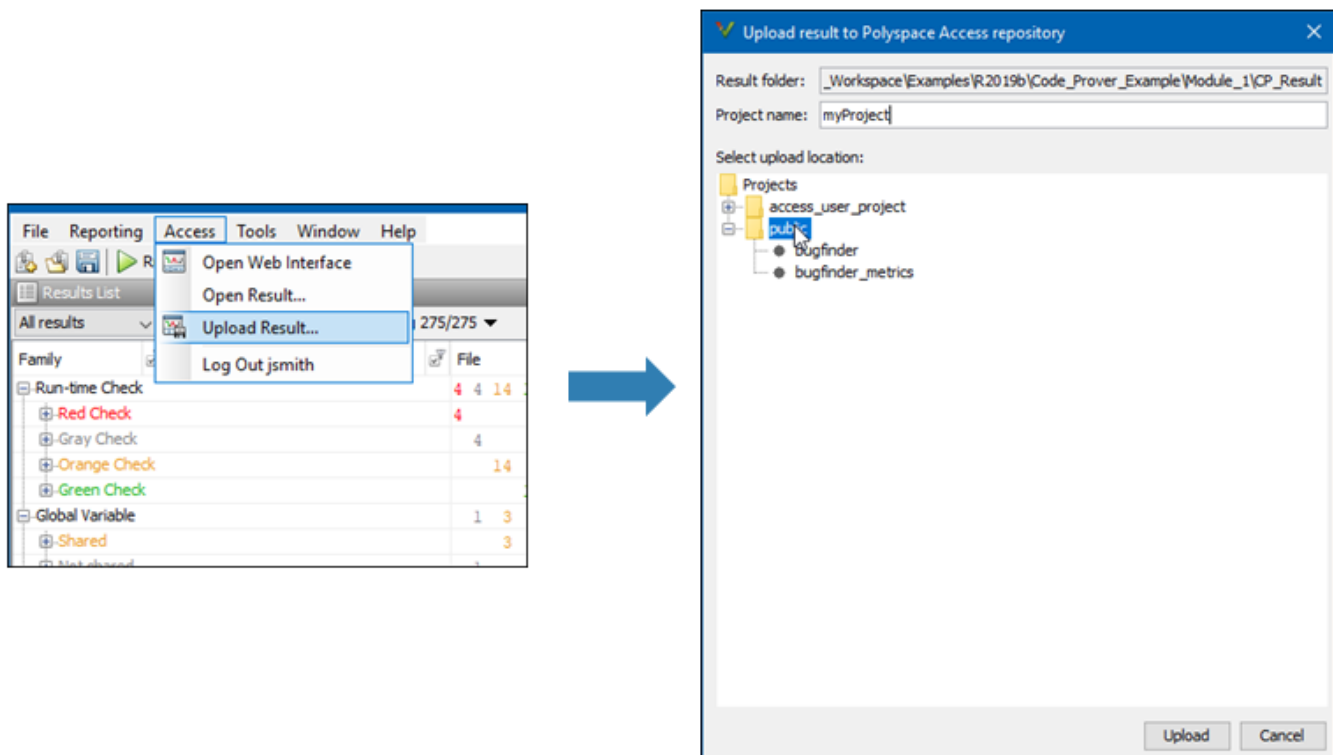
Upload Results to Polyspace Access

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

Upload Results from Polyspace Desktop Client

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

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



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

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

Once you save the changes you make to these fields in the desktop interface, the changes are reflected in the Polyspace Access web interface.

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

```
polyspace-access -host hostName -port port -upload BF_results
```

The command prompts you for your Polyspace Access login credentials, then uploads the results to the public folder of the Polyspace Access database. *hostName* is the fully qualified host name of the machine where you run the **Gateway** service. *port* is the **Port number** value specified for the **Gateway** service in the settings of the cluster operator. Depending on your configuration, you might also have to specify the `-protocol` option in the command. See “Configure Polyspace Access Services” (Polyspace Code Prover Access).

For additional information on `polyspace-access`, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server .

See Also

More About

- “Register Polyspace Desktop User Interface” (Polyspace Code Prover Access)
- “Interpret Results” (Polyspace Code Prover Access)
- “Manage Results” (Polyspace Code Prover Access)

Run Polyspace Analysis with Windows or Linux Scripts

- “Run Polyspace Analysis from Command Line” on page 2-2
- “Modularize Polyspace Analysis by Using Build Command” on page 2-4
- “polyspace-configure Source Files Selection Syntax” on page 2-10
- “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 2-12

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`

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
```

For the full list of analysis options, see:

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

More About

- “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 2-12
- “Modularize Polyspace Analysis by Using Build Command” on page 2-4

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\R2020a` or `C:\Program Files\Polyspace Server\R2020a`.

Build Source Code

Inspect the makefile. The makefile creates four binaries:


```
CC := gcc
LD := ld

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

all: cmd1 cmd2

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

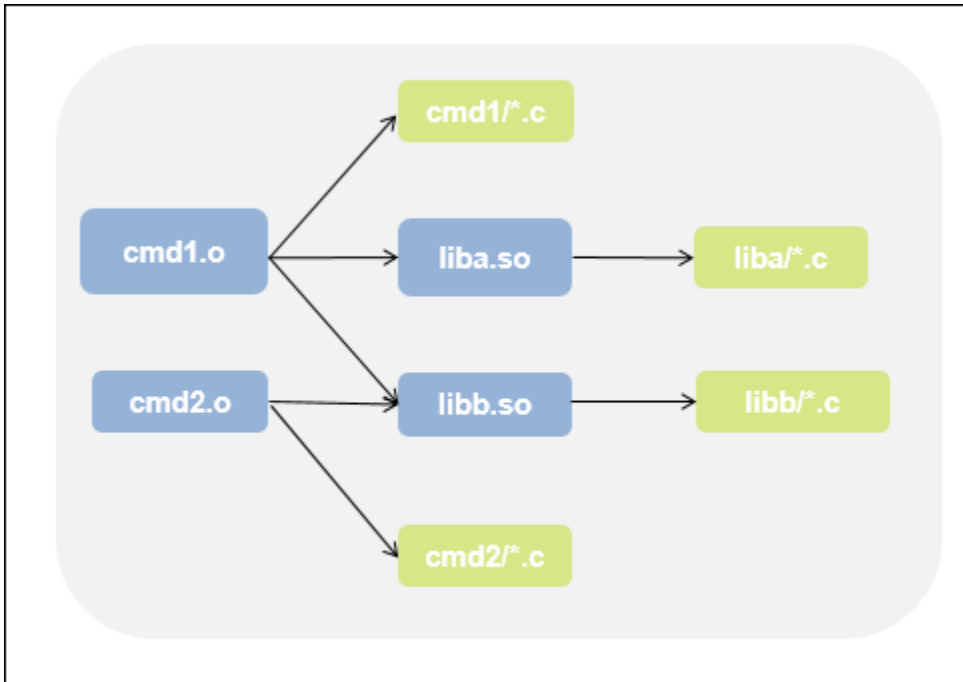
cmd2: libb
    $(CC) -c $(CMD2_SOURCES)
    $(LD) -o $@ $(CMD2_OBJ) $(LIBB_SOBJ)

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

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

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

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



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

```
make -B
```

Make sure that the build runs to completion.

Create One Polyspace Options File for Full Build

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

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

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

```
polyspace-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 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 14-6.

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 cmd1.psopts -results-dir results
```

For this approach, you do not need to know the details of your build command. However, when you create a separate options file for each binary in this way, each options file contains source files directly involved in the binary and not through shared objects. For instance, the options file `cmd1.psopts` in this example specifies only the source files in the `cmd1` subfolder and not the source files involved in creating the shared objects `liba.so` and `libb.so`. The subsequent analysis by using this options file cannot access functions from the shared objects and uses function stubs instead. In the Code Prover analysis, if you see too many orange checks due to the stubbing, use the approach stated in the section “Create Options File for Specific Binary in Build Command” on page 2-7.

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 14-6.

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

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

See Also

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

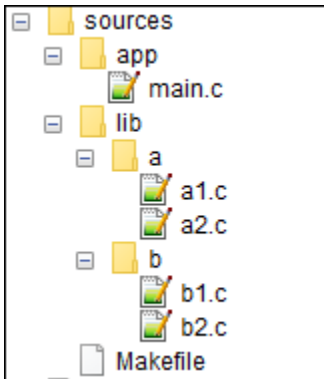
More About

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

polyspace-configure Source Files Selection Syntax

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

This folder structure applies to these examples.



To try these examples, use the demo files in `polyspaceroot\help\toolbox\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.

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

Glob Pattern Syntax	Example
No special characters, slashes ('/'), or backslashes ('\'). Pattern matches corresponding files, but not folders.	<code>-include-sources "main.c"</code> matches: <code>/sources/app/main.c</code>
Pattern contains '*' or '?' special characters. '*' matches zero or more characters in file or folder name. '?' matches one character in file or folder name.	<code>-include-sources "b?.c"</code> matches: <code>/sources/lib/b/b1.c</code> <code>/sources/lib/b/b2.c</code>
The matches do not include path separators.	<code>-include-sources "app/*.c"</code> matches: <code>/sources/app/main.c</code>

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

polyspace-configure does not support these glob patterns:

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

Configure Polyspace Analysis Options in User Interface and Generate Scripts

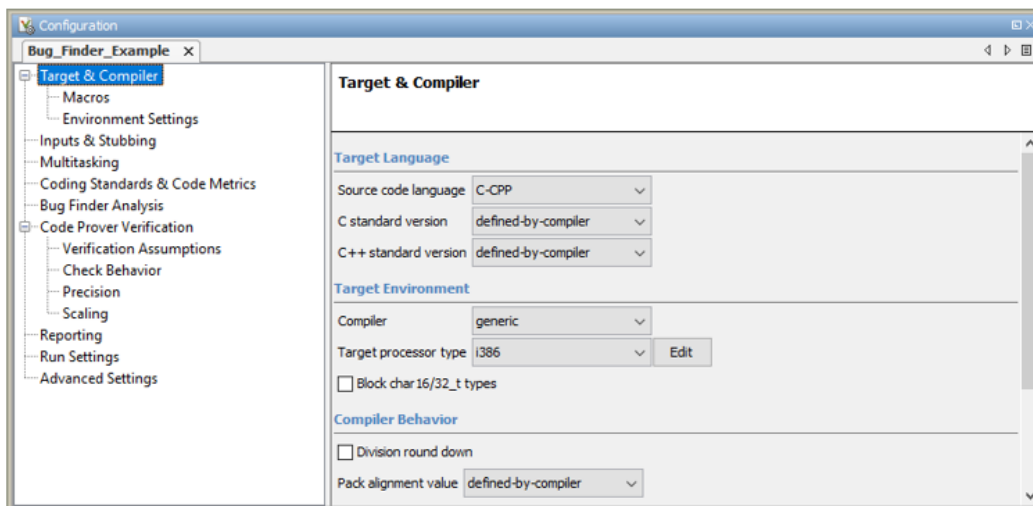
In this section...

“Prerequisites” on page 2-13

“Generate Scripts from Configuration” on page 2-13

“Run Analysis with Generated Scripts” on page 2-14

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



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

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

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

- Compilation assistant that suggests workarounds for some compilation errors,
- Auto-generation of XML file for constraint specification,
- Context-sensitive help for options.

Prerequisites

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

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

Generate Scripts from Configuration

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

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

Navigate to *polyspaceroot*\polyspace\bin, where *polyspaceroot* is the Polyspace installation folder; for instance, C:\Program Files\Polyspace\R2020a. Open the Polyspace user interface using the *polyspace* executable and create a new project.

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

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

See “Specify Polyspace Analysis Options” on page 8-2.

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

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

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

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

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

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

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

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

These files are generated for scripting the analysis:

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

Run Analysis with Generated Scripts

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

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

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

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

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

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

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

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

```
"C:\Program Files\Polyspace\R2020a\polyspace\bin\polyspace-code-prover.exe"
```

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

```
"C:\Program Files\Polyspace Server\R2020a\polyspace\bin\polyspace-code-prover-server.exe"
```

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

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

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

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

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

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

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

See Also

`-generate-launching-script-for`

Related Examples

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

Run Polyspace Analysis with MATLAB Scripts

- “Integrate Polyspace with MATLAB and Simulink” on page 3-2
- “Run Polyspace Analysis by Using MATLAB Scripts” on page 3-5
- “Compare Results from Different Polyspace Runs by Using MATLAB Scripts” on page 3-9
- “Generate MATLAB Scripts from Polyspace User Interface” on page 3-11
- “Troubleshoot Polyspace Analysis from MATLAB” on page 3-13

Integrate Polyspace with MATLAB and Simulink

Starting from R2019a, you can install Polyspace Bug Finder and Polyspace Code Prover as standalone products and analyze C/C++ code. However, you can interact with other MathWorks® products in these ways:

- Use MATLAB scripts to run Polyspace.

See “Polyspace Analysis with MATLAB Scripts”.

- Run Polyspace after C/C++ code generation from models in the Simulink Editor.

See “Polyspace Analysis in Simulink”.

- Run Polyspace after C/C++ code generation from MATLAB code in the MATLAB Coder™ App (if you have an Embedded Coder® license).

See “Polyspace Analysis in MATLAB Coder”.

If you install Polyspace products and other MathWorks products such as MATLAB, Simulink or Embedded Coder, you have to run the MATLAB installer twice and install Polyspace in a different root folder from the other products. For instance, in Windows:

- Your default MATLAB root folder is C:\Program Files\MATLAB\R2020a.
- Your default Polyspace root folder is C:\Program Files\Polyspace\R2020a.

To run Polyspace from within MATLAB, Simulink or MATLAB Coder, you have to perform a post-installation step to link your MATLAB and Polyspace installations.

Integrate Polyspace with MATLAB and Simulink Installation from Same Release

If your Polyspace and MATLAB installations belong to the same release, you can use all MATLAB functions and classes available for running Polyspace. You can also run Polyspace on generated code in the Simulink editor or MATLAB Coder App (with an Embedded Coder license).

To link your MATLAB and Polyspace installations:

- 1 Open MATLAB with administrator privileges.
- 2 Navigate to *polyspaceroot*\toolbox\polyspace\pscore\pscore\. Here, *polyspaceroot* is the Polyspace installation folder, for instance, C:\Program Files\Polyspace\R2020a.
- 3 At the MATLAB command prompt, enter:

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

You see a prompt stating that the workspace will be cleared and all open models closed. Click **Yes** to continue the linking. The process might take a few minutes to complete.

To avoid the prompt during installation, enter:

```
polyspacesetup('install', 'silent', true)
```

- 4 Restart MATLAB. You can now use all functions and classes available for running Polyspace.

A MATLAB installation can be linked with only one Polyspace installation. To link to a new Polyspace installation, any previous links must be removed. To remove a link between a Polyspace and MATLAB installation, at the MATLAB command prompt, enter:

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

Integrate Polyspace with Simulink Installation from Different Release

You can link your Polyspace installation with an earlier release of Simulink. The Simulink release can be at most 4 releases behind the Polyspace release and must not already have Polyspace installed in the same folder. For instance, you can link the release R2017b of Simulink with the release R2019a of Polyspace provided the release R2017b does not already have Polyspace in it. The only exception is that release R2020a of Polyspace cannot be linked specifically with the release R2019b of Simulink. To check if Polyspace is already installed, see “Check Integration Between MATLAB and Polyspace” on page 3-4.

If you link a Polyspace and Simulink installation from different releases, you can use only a subset of MATLAB functions that apply to Simulink.

To link an R2017b release of Simulink with an R2019a release of Polyspace, in your post-installation step:

- 1 Open MATLAB from the earlier release (R2017b) with administrator privileges.
- 2 Navigate to *polyspaceroot*\toolbox\polyspace\pscore\pscore\. Here, *polyspaceroot* is the installation folder of the later Polyspace release (R2019a).
- 3 At the MATLAB command prompt, enter:

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

You see a prompt stating that the workspace will be cleared and all open models closed. Click **Yes** to continue the linking.

To avoid prompts during installation, enter:

```
polyspacesetup('install', 'silent', true)
```

- 4 Restart MATLAB.

If you link a Polyspace installation with an earlier release of Simulink, you cannot use all functions and classes available to run a Polyspace analysis. You can only:

- Use the functions `pslinkoptions`, `pslinkrun` and `pslinkfun` to run Polyspace on generated code.
- Run Polyspace on C/C++ code generated from models in the Simulink editor.

You can only verify generated code. You cannot verify handwritten C/C++ code in S-Functions and C Caller blocks.

In addition, you cannot:

- Use the `polyspace.Project`, `polyspace.Options` and `polyspace.ModelLinkOptions` classes or the `polyspaceCodeProver` and `polyspaceBugFinder` functions.

- Run Polyspace on C/C++ code generated from MATLAB code in the MATLAB Coder App.

Check Integration Between MATLAB and Polyspace

To check if a MATLAB installation is already linked to a Polyspace installation, open MATLAB and enter:

```
ver
```

You see the list of products installed. If Polyspace is linked to MATLAB (after R2019a) or in the same installation folder as MATLAB (prior to R2019a), 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')
```

Prior to R2019a, if Polyspace is installed in the same folder as MATLAB, to cross-link to a newer version of Polyspace without upgrading MATLAB, you have to uninstall the older Polyspace using the MATLAB uninstaller. After R2019a, you can simply remove the link between a Polyspace and MATLAB installation. To remove an existing link between MATLAB and Polyspace, at the MATLAB command prompt, enter:

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

See Also

More About

- “Polyspace Analysis with MATLAB Scripts”
- “Polyspace Analysis in Simulink”
- “Polyspace Analysis in MATLAB Coder”

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
bfSummary = proj.Results.getSummary('defects');
```

See also `polyspace.Project`.

Prerequisites

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

Specify Multiple Source Files

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

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

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

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

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

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

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

You can explicitly exclude files from analysis. For instance:

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

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

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

Check for MISRA C:2012 Violations

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

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

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

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

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

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

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

Check for Specific Defects or Coding Rule Violations

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

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

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

misraRules.Section_8_Declarations_and_definitions.rule_8_1 = false;
misraRules.Section_8_Declarations_and_definitions.rule_8_2 = false;
misraRules.Section_8_Declarations_and_definitions.rule_8_3 = false;
```

```
misraRules.Section_8_Declarations_and_definitions.rule_8_4 = false;

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

See also `polyspace.CodingRulesOptions`.

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

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

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

Find Files That Do Not Compile

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

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

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

However, it is more convenient to let the analysis complete and capture all compilation errors from the analysis log file. For more information, see “Troubleshoot Polyspace Analysis from MATLAB” on page 3-13.

Run Analysis on Server

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

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

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

Run analysis as usual.

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

Open the results from the results folder location.

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

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

See Also

`polyspace.Project` | `polyspaceCodeProver`

Related Examples

- “Generate MATLAB Scripts from Polyspace User Interface” on page 3-11
- “Visualize Code Prover Analysis Results in MATLAB” on page 20-13
- “Troubleshoot Polyspace Analysis from MATLAB” on page 3-13

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

with this section:

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

See Also

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

More About

- “Run Polyspace Analysis by Using MATLAB Scripts” on page 3-5

Generate MATLAB Scripts from Polyspace User Interface

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

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

Prerequisites

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

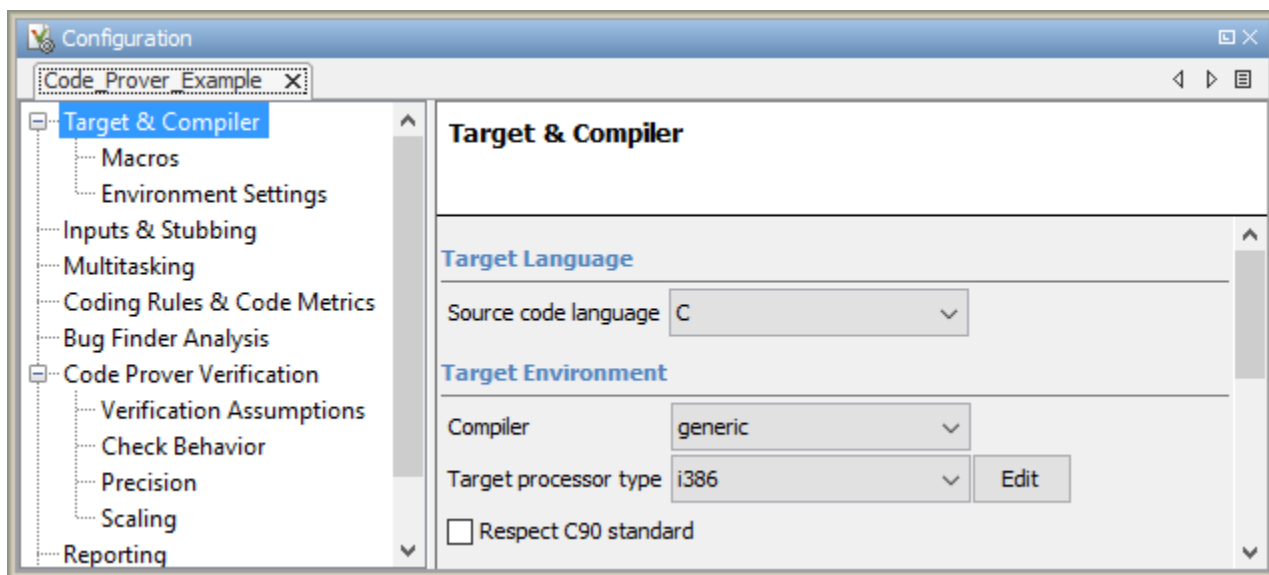
Create Scripts from Polyspace Projects

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

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

From this project, you can generate a script that contains your sources, includes and other analysis options. To begin, select **File > New Project**. For details, see “Add Source Files for Analysis in Polyspace User Interface” on page 1-2.

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



- 1 Open MATLAB.

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

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

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

- 3 Append the object to a MATLAB script.

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

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

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

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

```
run(filePath);
```

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

See Also

Related Examples

- “Run Polyspace Analysis by Using MATLAB Scripts” on page 3-5

Troubleshoot Polyspace Analysis from MATLAB

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

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

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

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

Prerequisites

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

Capture Polyspace Analysis Errors in Error Log

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

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

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

The function is defined as follows.

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

    proj = polyspace.Project;

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

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

    % Run analysis
```

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

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

% Check log file for compilation errors
numErrors = 0;

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

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

fclose(errorFile);

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

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

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

See Also

`polyspace.Project`

Related Examples

- “Run Polyspace Analysis by Using MATLAB Scripts” on page 3-5
- “Troubleshoot Compilation Errors”

Offload Polyspace Analysis to Remote Servers from Desktop

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

Send Polyspace Analysis from Desktop to Remote Servers

In this section...

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

“Prerequisites” on page 4-3

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

You can perform a Polyspace analysis locally on your desktop or offload the analysis to one or more dedicated servers. You offload a Polyspace analysis from a Polyspace desktop product such as Polyspace Bug Finder but the analysis runs on the server using a Polyspace server product such as Polyspace Bug Finder Server.

This topic shows how to send a Polyspace analysis from the user interface of the Polyspace desktop products.

- To offload an analysis with scripts, see “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts” on page 4-6.
- For a simple tutorial that walks through all the steps for offloading a Polyspace analysis, see “Send 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.

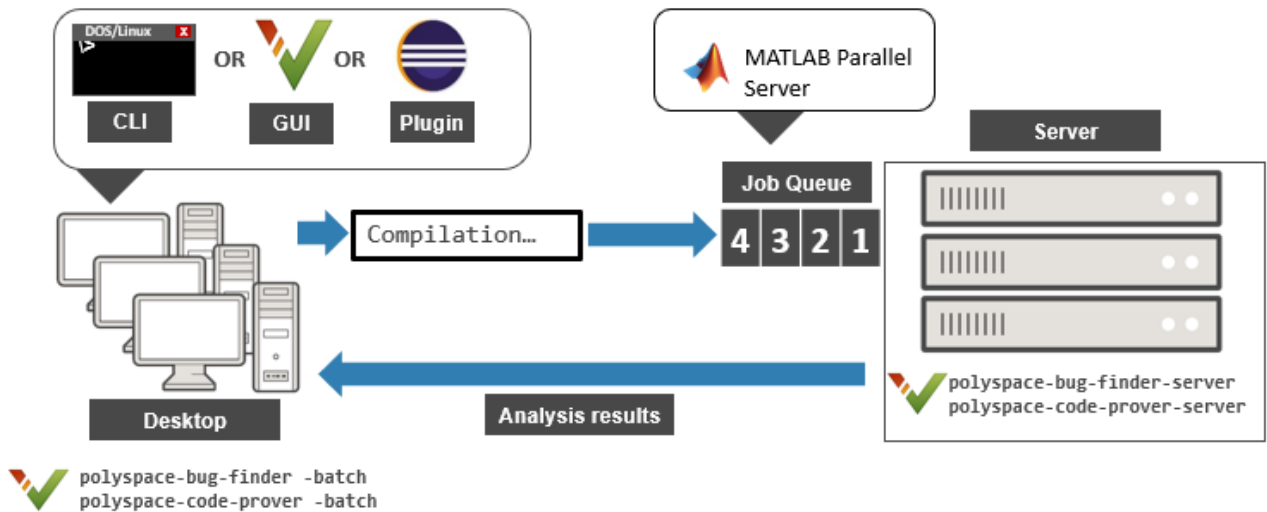
You require the Polyspace desktop product, Polyspace Bug Finder on the computer that acts as the client node.

- 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.



Prerequisites

Before offloading an analysis from the user interface of the Polyspace desktop products, you must set up your project's source files, analysis options, and remote analysis settings. If you have not done so, for more information on:

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

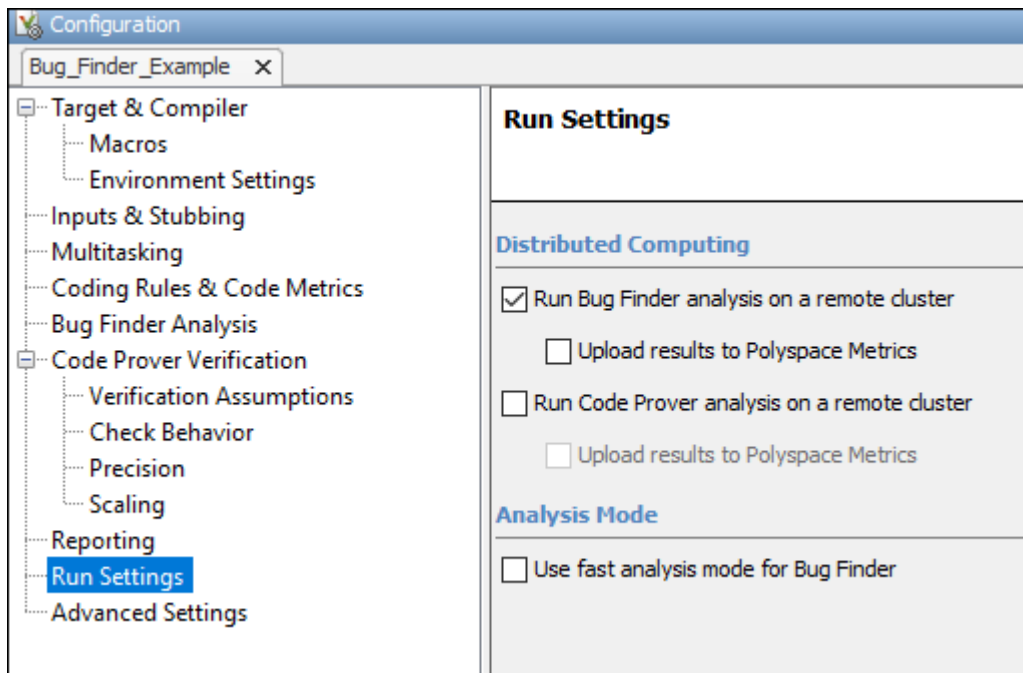
Once you have set up a Polyspace project and established communication between a desktop and a remote server, you are ready to offload a Polyspace analysis.

Offload Analysis in Polyspace User Interface

To start a remote analysis:

- 1 Select a project to analyze.
- 2 On the **Configuration** pane, select **Run Settings**.

Select **Run Bug Finder analysis on a remote cluster** and/or **Run Code Prover analysis on a remote cluster**.



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

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

- 4 Start the analysis. For instance, to start a Bug Finder analysis, click the **Run Bug Finder** button.

The compilation part of the analysis takes place on the desktop product. After compilation, the analysis is offloaded to the server.

- 5 To monitor the analysis, select **Tools > Open Job Monitor**. In the Polyspace Job Monitor, follow your queued job to monitor progress.

Once the analysis is complete, the results are downloaded back to the user interface of the Polyspacedesktop products. You can open the results directly in the user interface. If you uploaded the results to Polyspace Metrics, you have to explicitly download them from the Polyspace Metrics interface.

If the analysis stops after compilation and you have to restart the analysis, to avoid restarting from the compilation phase, use the option `-submit-job-from-previous-compilation-results`.

Note If you choose to upload results to Polyspace Metrics, your results are not downloaded automatically after verification. Use the Polyspace Metrics web dashboard to view the results and download them to your desktop. For more information, see “View Code Quality Metrics” on page 21-12.

See Also

Run Bug Finder or Code Prover analysis on a remote cluster (-batch)

More About

- “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”
- “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts” on page 4-6

Send Polyspace Analysis from Desktop to Remote Servers Using Scripts

Instead of running a Polyspace analysis on your local desktop, you can send the analysis to a remote cluster. You can use a dedicated cluster for running Polyspace to free up memory on your local desktop.

This topic shows how to use Windows or Linux scripts to send the analysis to a remote cluster and download the results to your desktop after analysis.

- To offload an analysis from the Polyspace user interface, see “Send Polyspace Analysis from Desktop to Remote Servers” on page 4-2.
- For a simple tutorial that walks through all the steps for offloading a Polyspace analysis, see “Send 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.

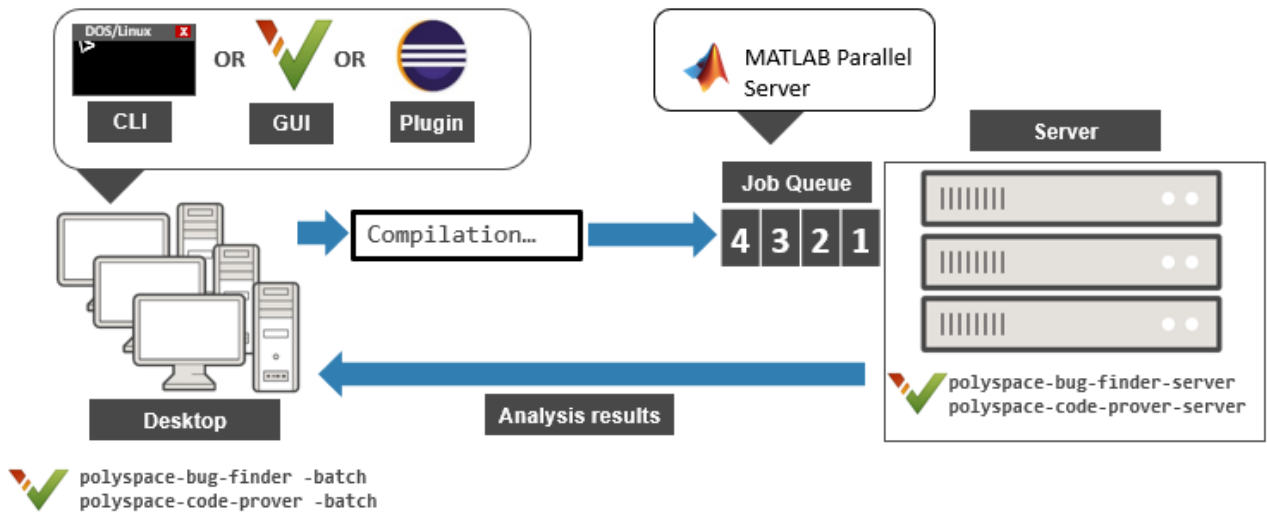
You require the Polyspace desktop product, Polyspace Bug Finder on the computer that acts as the client node.

- 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.



Prerequisites

Before you run a remote analysis by using scripts, you must set up communication between a desktop and a remote server. See “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

Run Remote Analysis

To run a remote analysis, use this command:

```

polyspaceroot\polyspace\bin\polyspace-code-prover
  -batch -scheduler NodeHost|MJSName@NodeHost [options] [-mjs-username name -password pswd]

```

where:

- *polyspaceroot* is the installation folder of Polyspace desktop products, for instance, C:\Program Files\Polyspace\R2020a.
- *NodeHost* is the name of the computer that hosts the head node of the MATLAB Parallel Server cluster.

MJSName is the name of the MATLAB Job Scheduler on the head node host.

If you set up communications with a cluster from the Polyspace user interface, you can determine *NodeHost* and *MJSName* from the user interface. Select **Metrics > Metrics and Remote Server Settings**. Open the MATLAB Parallel Server Admin Center. Under **MATLAB Job Scheduler**, see the **Name** and **Hostname** columns for *MJSName* and *NodeHost*.

If you use the `startjobmanager` command to start the MATLAB Job Scheduler, *MJSName* is the argument of the option `-name`. For details, see “Configure Advanced Options for MATLAB Job Scheduler Integration” (MATLAB Parallel Server).

- *options* are the analysis options. These options are the same as that of a local analysis. For instance, you can use these options:
 - `-sources-list-file`: Specify a text file with one source file name per line.
 - `-options-file`: Specify a text file with one option per line.
 - `-results-dir`: Specify a download folder for storing results after analysis.

For the full list of options, see “Analysis Options”. Alternatively, you can:

- Start an analysis in the user interface and stop after compilation. You can obtain the text files and scripts for running the analysis at the command line. See “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 2-12.
- 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* and *pswd* are the username and password required for job submissions using MATLAB Parallel Server. These credentials are required only if you use a security level of 2 or higher for MATLAB Parallel Server submissions. See “Set MATLAB Job Scheduler Cluster Security” (MATLAB Parallel Server).

The analysis executes locally on your desktop up to the end of the compilation phase. After compilation, the software submits the analysis job to the cluster and provides a job ID. You can also read the ID from the file `ID.txt` in the results folder. To monitor your analysis, use the `polyspace-jobs-manager` command with the job ID.

If the analysis stops after compilation and you have to restart the analysis, to avoid rerunning the compilation phase, use the option `-submit-job-from-previous-compilation-results`.

Manage Remote Analysis

To manage multiple remote analyses, use the option `-batch`. For instance:

```
polyspaceroot\polyspace\bin\polyspace-jobs-manager action  
-scheduler schedulerName
```

See also Run Bug Finder or Code Prover analysis on a remote cluster (`-batch`). Here:

- *polyspaceroot* is your MATLAB installation folder.
- *schedulerName* is one of the following:
 - Name of the computer that hosts the head node of your MATLAB Parallel Server cluster (*NodeHost*).
 - Name of the MATLAB Job Scheduler on the head node host (*MJSName@NodeHost*).

- Name of a MATLAB cluster profile (*ClusterProfile*).

For more information about clusters, see “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox)

If you do not specify a job scheduler, `polyspace-job-manager` uses the scheduler specified in the Polyspace preferences. To see the scheduler name, select **Tools > Preferences**. On the **Server Configuration** tab, see the **Job scheduler host name**.

- `action` refers to the possible action commands to manage jobs on the scheduler:

- `listjobs`:

Generate a list of Polyspace jobs on the scheduler. For each job, the software produces this information:

- `ID` — Verification or analysis identifier.
- `AUTHOR` — Name of user that submitted job.
- `APPLICATION` — Name of Polyspace product, for example, Polyspace Code Prover or Polyspace Bug Finder.
- `LOCAL_RESULTS_DIR` — Results folder on local computer, specified through the **Tools > Preferences > Server Configuration** tab.
- `WORKER` — Local computer from which job was submitted.
- `STATUS` — Status of job, for example, `running` and `completed`.
- `DATE` — Date on which job was submitted.
- `LANG` — Language of submitted source code.
- `download -job ID -results-folder FolderPath`:

Download results of analysis with specified ID to folder specified by *FolderPath*.

When the analysis job is queued on the server, the command `polyspace-code-prover` returns a job id. In addition, a file `ID.txt` in the results folder contains the job ID in this format:

```
job_id;server_name:project_name version_number
```

For instance, `92;localhost:Demo 1.0`.

If you do not use the `-results-folder` option, the software downloads the result to the folder that you specified when starting analysis, using the `-results-dir` option.

After downloading results, use the Polyspace user interface to view the results.

- `getlog -job ID`:

Open log for job with specified ID.

- `remove -job ID`:

Remove job with specified ID.

- `promote -job ID`:

Promote job with specified ID in the queue.

- `demote -job ID`

Demote job with specified ID in the queue.

Sample Scripts for Remote Analysis

In Windows, to avoid typing the commands each time, you can save the commands in a batch file. In Linux, you can relaunch the analysis by using a shell script. To create a batch file for running analysis:

- 1 Save your analysis options in a file `listoptions.txt`. See `-options-file`.
- 2 Create a file `launcher.bat` in a text editor like Notepad.

In the file, enter these commands:

```
echo off
set POLYSPACE_PATH=polyspaceroot\polyspace\bin
set RESULTS_PATH=C:\Results
set OPTIONS_FILE=C:\Options\listoptions.txt
"%POLYSPACE_PATH%\polyspace-code-prover.exe" -batch -scheduler localhost
                                     -results-dir %RESULTS_PATH% -options-file %OPTIONS_FILE%
pause
```

polyspaceroot is the Polyspace installation folder. *localhost* is the name of the computer that hosts the head node of your MATLAB Parallel Server cluster.

- 3 Replace the definitions of these variables in the file:
 - `POLYSPACE_PATH`: Enter the actual location of the `.exe` file.
 - `RESULTS_PATH`: Enter the path to a folder. The files generated during compilation are saved in the folder.
 - `OPTIONS_FILE`: Enter the path to the file `listoptions.txt`.
- 4 Double-click `launcher.bat` to run the analysis.

Tip If you run a Polyspace analysis, a Windows `.bat` or Linux `.sh` file is generated. The file is in the `.settings` subfolder in your results folder. Instead of writing a script from scratch, you can relaunch the analysis using this file.

See Also

Run Bug Finder or Code Prover analysis on a remote cluster (`-batch`)

More About

- “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”
- “Send Polyspace Analysis from Desktop to Remote Servers” on page 4-2

Run Polyspace Analysis on Generated Code

Run Polyspace Analysis on Code Generated with Embedded Coder

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

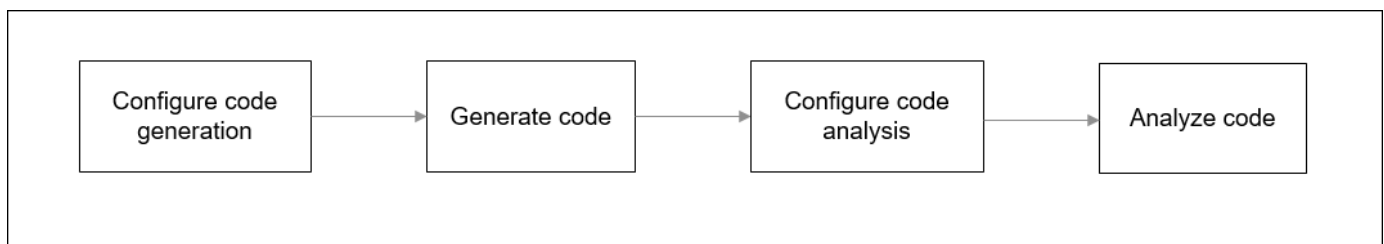
This topic uses Embedded Coder for code generation. For analysis of TargetLink-generated code, see “Run Polyspace Analysis on Code Generated with TargetLink” on page 5-49.

For a tutorial with a specific model, see “Run Polyspace Analysis on Code Generated from Simulink Model” on page 5-10.

Prerequisites

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

Generate and Analyze Code



Configure Code Generation and Generate Code

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

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

- “Code-to-model” (Embedded Coder): Select this option to enable links from code to model.

For the full list of parameters to set, see “Recommended Model Configuration Parameters for Polyspace Analysis” on page 5-37.

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

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

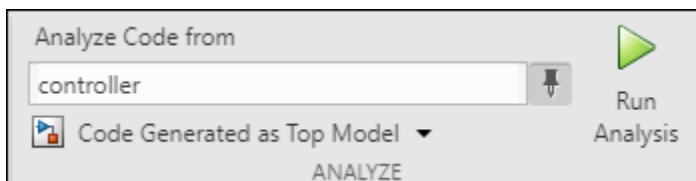
Configure Code Analysis

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

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

Analyze Code

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



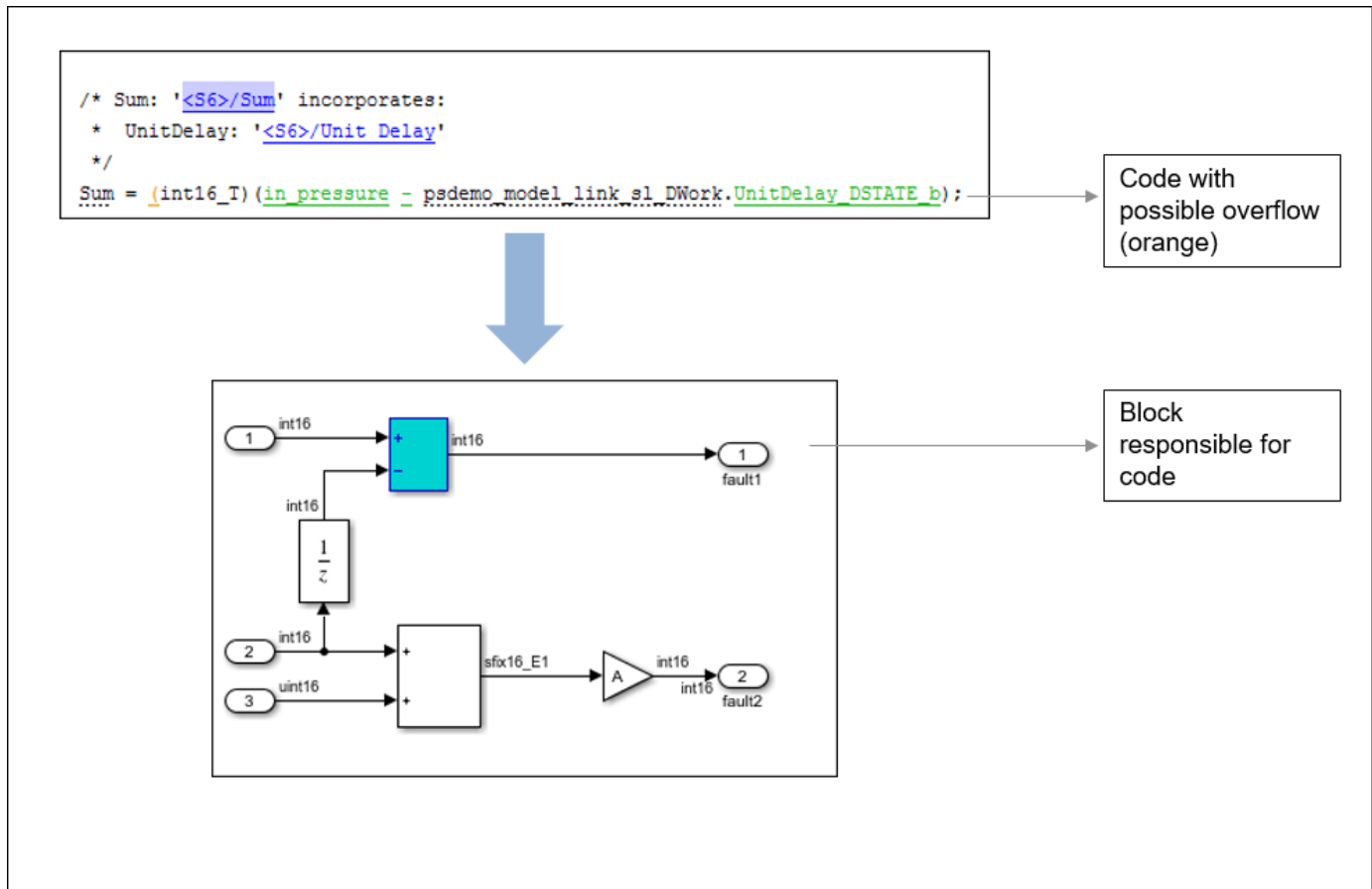
If the current model is referenced in another model and you want to verify the generated code in the context where the model is referenced, instead of **Code Generated as Top Model**, use **Code Generated as Model Reference**.

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 Polyspace Code Prover Results" on page 16-2
- "Code Prover Result and Source Code Colors" on page 16-8
- "Address Polyspace Results Through Bug Fixes or Justifications" on page 18-2
- "Filter and Group Results" on page 19-2

Navigate from Code to Model

Links in code comments show blocks that generate the subsequent lines of code. To see the blocks in the model, click the block names in the links. If you encounter issues, see “Troubleshoot Navigation from Code to Model” on page 5-54.

Alternatively, you can right-click a variable name and select **Go to Model**. This option is not available for all variables.

Fix Issue

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

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

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

If you include handwritten C/C++ code in S-function blocks, the Polyspace analysis can reveal possible integration issues between the handwritten and generated code. You can also analyze the handwritten code in isolation. See “Run Polyspace Analysis on S-Function Code” on page 5-21.

Annotate Blocks to Justify Issues

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

To annotate a block, select the block and on the **Polyspace** tab, select **Add Annotation**. Enter the following:

- Comma-separated list of result acronyms. To justify only the type of result, select **Only 1 check**.

See:

- “Short Names of Bug Finder Defect Checkers” (Polyspace Bug Finder)
- “Short Names of Code Prover Run-Time Checks” on page 18-12
- Status, severity, and comment to be assigned to the results.

Sometimes operations in the generated code are known to cause orange checks in Code Prover. Suppose an operation is known to possibly overflow. 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 automatically justify these checks through code comments, specify the configuration parameter “Operator annotations” (Embedded Coder).

When you reuse an annotated block in a different position or model, the changed context can render the annotation incorrect. To avoid incorrect annotations:

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

See Also

More About

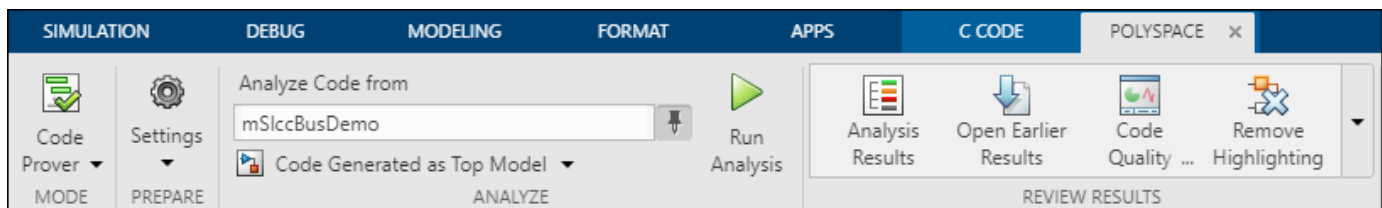
- “Configure Advanced Polyspace Options in Simulink” on page 5-39

Changes in Polyspace Analysis Workflows in Simulink in R2019b

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

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

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



Code Verification Workflow in a Nutshell

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

- 1 Select product to run:

For instance, select **Bug Finder**.

- 2 Specify code analysis options:

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

- 3 Specify which code to analyze:

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

- 4 Run analysis:

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

Locate Pre-R2019b Menu Items in Simulink Toolstrip

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

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

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

See Also

More About

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

Run Polyspace Analysis on Code Generated from Simulink Model

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

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

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

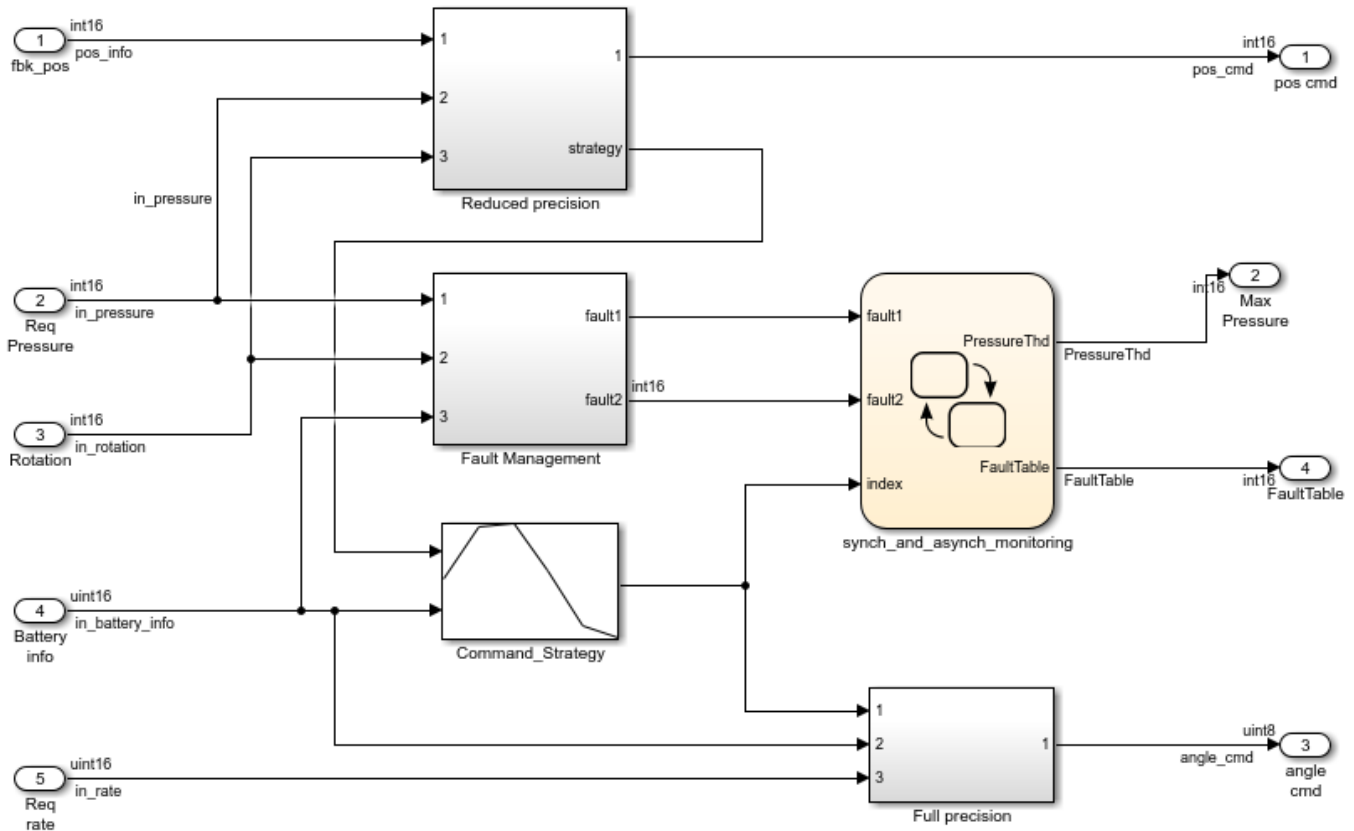
Prerequisites

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

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

Open Model for Code Generation and Polyspace Analysis

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



Generate Code

To generate code from the model, on the **Apps** tab, select **Embedded Coder**. On the **C Code** tab, select **Generate Code**.

Equivalent MATLAB Code:

```
load_system('polyspace_controller_demo');
rtwbuild('polyspace_controller_demo');
```

Analyze Code

To analyze the code generated from the model, on the **Apps** tab, select **Polyspace Code Verifier**. On the **Polyspace** tab:

- 1 Select the product to run: **Bug Finder** or **Code Prover**.
- 2 Click anywhere on the canvas. The **Analyze Code from** field shows the model name. Select **Run Analysis**.

Equivalent MATLAB Code:

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

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

Review Analysis Results

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

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

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

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

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

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

Trace Errors Back to Model and Fix Them

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

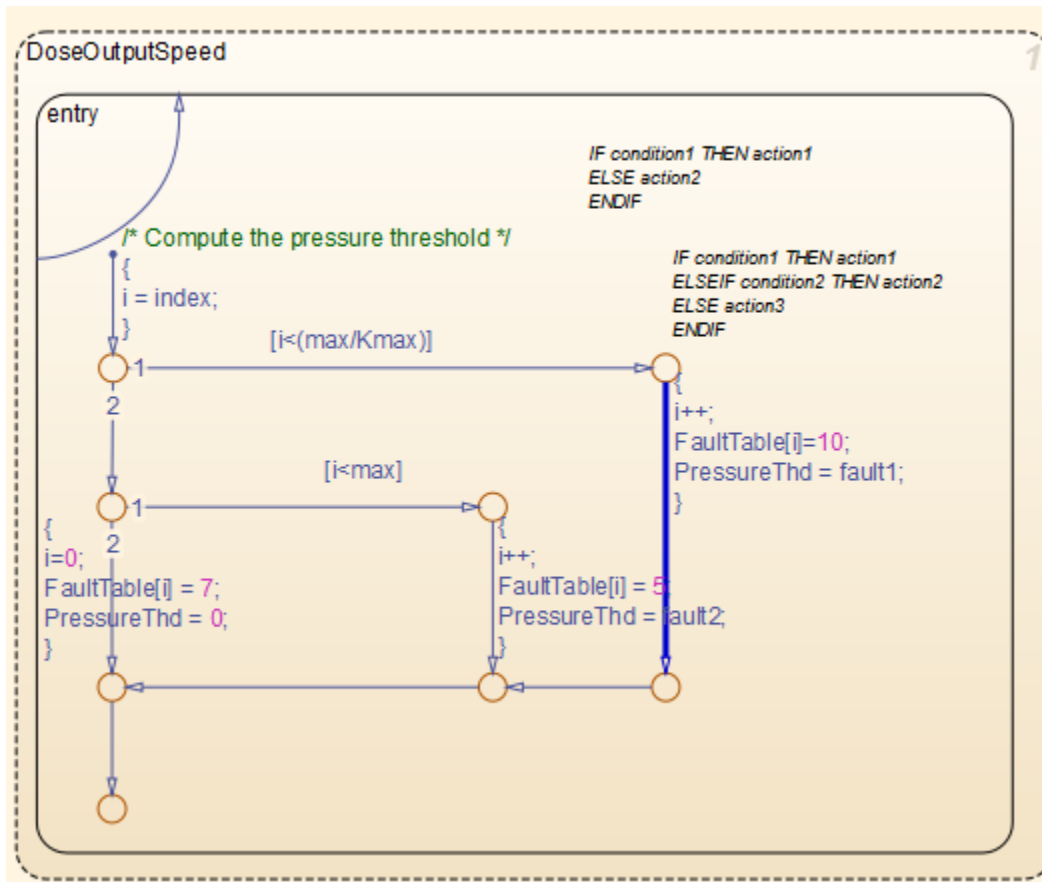
Error 1: Out of bounds array index

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

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

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

You see that the error occurs due to a transition in the Stateflow chart `synch_and_async_monitoring`. You can trace the error to the input variable `index` of the Stateflow chart.



You can avoid the **Out of bounds array index** in several ways. One way is to constrain the input variable `index` using a Saturation block before the Stateflow chart.

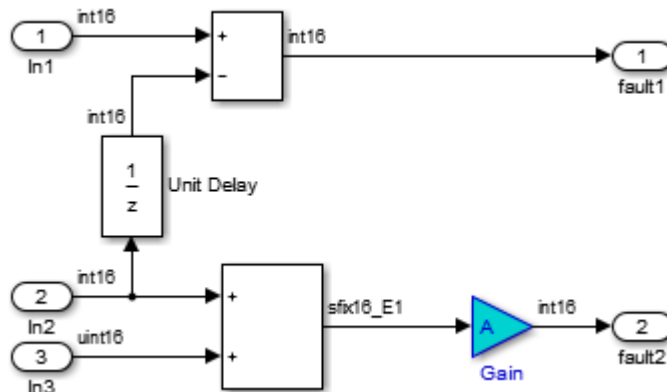
Error 2: Overflow

- 1 On the **Results List** pane, select the orange **Overflow** error shown below. The error appears in the file `polyspace_controller_demo.c`.
- 2 On the **Source** pane, review the error. To trace the error back to the model, click the link **S1/Gain** in comments above the orange error.

```
/* Gain: '<S1>/Gain' incorporates:
 * Inport: '<Root>/Battery Info'
 * Inport: '<Root>/Rotation'
 * Sum: '<S1>/Sum1'
 */
```

```
Gain = (int16_T)((((int16_T)((in_rotation + in_battery_info) >> 1) * 24576) >>
10);
```

You see that the error occurs in the Fault Management subsystem inside a Gain block following a Sum block.



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

- 1 Double-click the Inport block `Battery info` that provides the input signal `in_battery_info` to the model.
- 2 On the **Signal Attributes** tab, change the **Maximum** value of the signal.

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

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

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

Check for Coding Rule Violations

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

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

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

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

Annotate Blocks to Justify Results

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

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

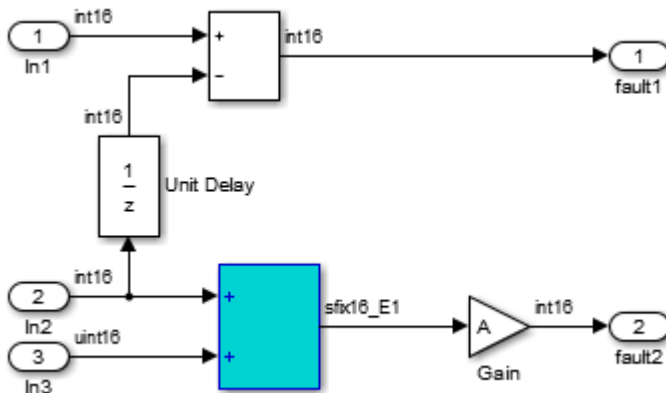
- 2 In the file `polyspace_controller_demo.c`, in the function `polyspace_controller_demo_step()`, select the violation of MISRA C:2012 rule 10.4. The **Source** pane shows that an addition operation violates the rule.
- 3 On the **Source** pane, click the link **S1/Sum1** in comments above the addition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

```

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

```

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



To annotate this block and justify the rule violation:

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

See Also

More About

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

Fix Model Design Issues Found as Bugs 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.

Following are examples of issues that are detected in the generated code but can be fixed in the original model.

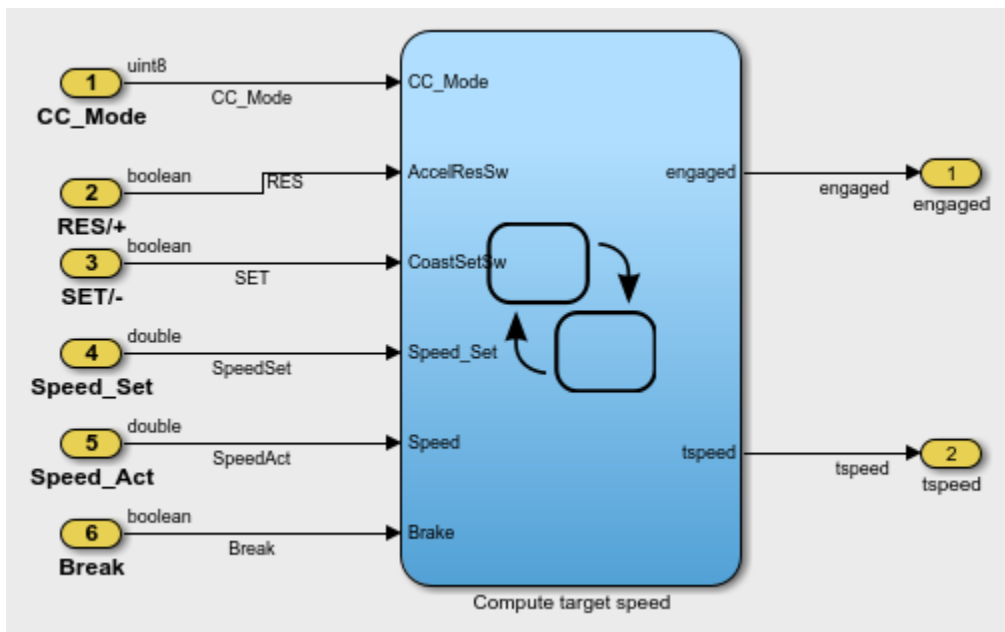
Prerequisites

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

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

Open Model with Stateflow Chart Containing Design Issues

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.



Generate and Analyze Code

Generate C code from the model and check the generated code for run-time errors by using Polyspace Code Prover.

Generate Code

To generate code, on the **Apps** tab, select **Embedded Coder**. Then, on the **C Code** tab, select **Generate Code**. Follow the progress of code generation in the Simulink Diagnostic Viewer.

For more information, see “Generate C Code from Simulink Models” (Embedded Coder).

Configure Code Analysis

A default Code Prover analysis runs Code Prover run-time checks only. Enable MISRA C:2012 checking in addition to the default checks.

On the **Polyspace** tab, select **Settings** to open the Simulink Configuration Parameters window. In the **Settings from** dropdown, select Project configuration and MISRA C 2012 checking.

Check Code for Errors

On the **Polyspace** tab, click anywhere on the canvas. The **Analyze Code from** field shows the model name. Select **Run Analysis**. Follow the progress of code analysis in the MATLAB Command Window.

For more information, see “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 5-2.

Fix Issues

The Code Prover results open in the Polyspace user interface. The results contain some 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.

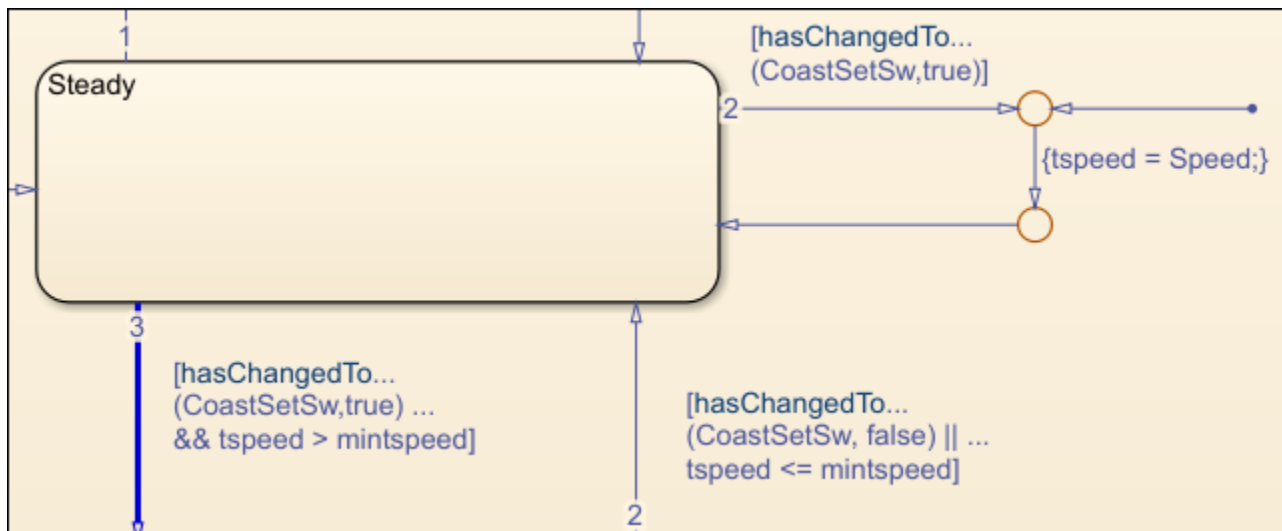
```

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.

You can fix the issue in various ways. One possibility is to switch the execution order of transitions 2 and 3. To begin, right-click transition 3.

If you 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 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:

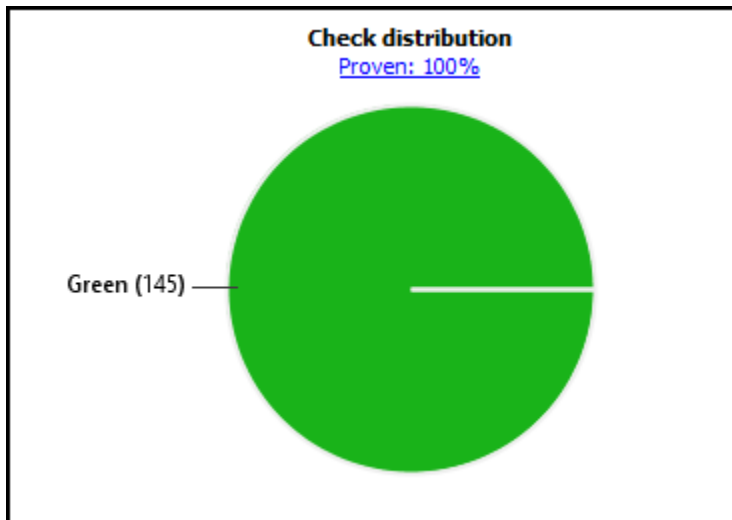
In the Model Explorer window, change the storage class of `holdrate` from `Global` to `Define`. The generated code defines a typedef that ensures that `holdrate` has the value 5.

```
#define holdrate 5
```

- Modify analysis configuration:

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

If you regenerate and reanalyze the code, you no longer see the orange **Division by zero** checks. You also do not see the other orange checks because they have the same root cause. The **Dashboard** pane shows that all checks are green.



Fix MISRA C:2012 Violations

Select the violation of rule 3.1:

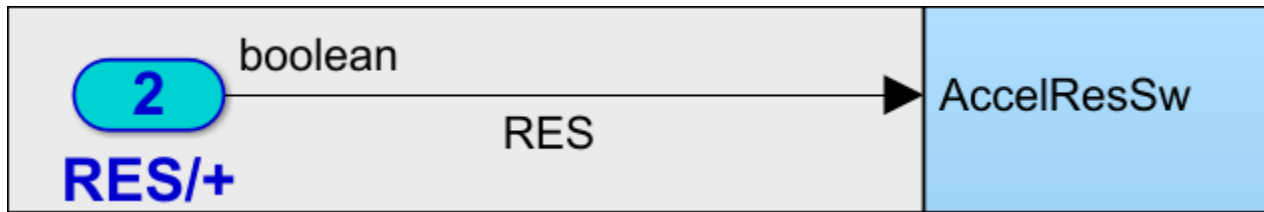
The character sequences `/*` and `//` shall not be used within a comment.

Review the code.

```
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;
```

You see two instances of `//` in 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 variable and also the variable 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 5-10

Run Polyspace Analysis on S-Function Code

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

Prerequisites

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

S-Function Analysis Workflow

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

- 1 Compile your S-function to be compatible with Polyspace.
- 2 Select your Polyspace options.
- 3 Run a Polyspace 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” on page 19-2.
 - For help reviewing and understanding the results, see “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 `-sl dv`.

Example S-Function Analysis

This example shows the workflow for analyzing S-functions with Polyspace. You use the model `psdemo_model_link_sl` and the S-function `Command_Strategy`.

- 1 Open the model and use the Legacy Code Tool to compile the S-function `Command_Strategy`.

```
% Open Model
psdemo_model_link_sl
```

```
% Compile S-function Command_Strategy
def = legacy_code('initialize');
def.SourceFiles = { 'command_strategy_file.c' };
def.HeaderFiles = { 'command_strategy_file.h' };
def.SFunctionName = 'Command_Strategy';
def.OutputFcnSpec = 'int16 y1 = command_strategy(uint16 u1, uint16 u2)';
def.IncPaths = { fullfile(polyspaceroot, ...
    'toolbox','polyspace','pslink','pslinkdemos','psdemo_model_link_sl') };
def.SrcPaths = def.IncPaths;
def.Options.supportCoverageAndDesignVerifier = true;
legacy_code('compile',def);
```

- 2 Open the model `psdemo_model_link_sl/controller`.
- 3 Specify the code analysis options. On the **Apps** tab, select **Polyspace Code Verifier**. Then, on the **Polyspace** tab:

- Select the product to run: **Bug Finder** or **Code Prover**.
- Select **Settings**. In the Configuration Parameters dialog box, make sure that the following parameters are set:
 - **Settings from** — Project configuration and MISRA C 2012 checking
 - **Open results automatically after verification** — On

Apply your settings and close the Configuration Parameters.

- 4 Right-click the `Command_Strategy` block and select **Polyspace > Verify S-Function > This Occurrence**.
- 5 Follow the analysis in the MATLAB Command Window. When the analysis is finished, your results open in the Polyspace interface.

See Also

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 5-2
- “Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts” on page 5-23

Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts

You can check for bugs and run-time errors in the custom C/C++ code used in your Simulink model. The Polyspace analysis checks functions called from C Caller blocks and Stateflow charts with inputs from the model.

Prerequisites

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

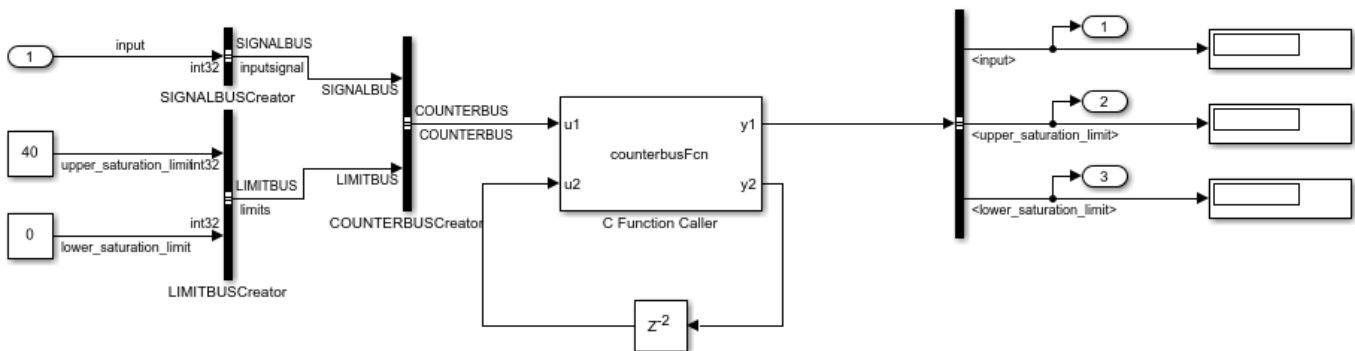
To open the models used in this example, look for this example in the MATLAB Help browser and click the **Open Model** buttons.

C/C++ Function Called Once in Model

This example uses a function called only once in the model from a C Caller block. The analysis checks the function using inputs to the C Caller block.

Open Model for Running Analysis on Custom Code

Open the model `mSlcCbBusDemo` for analyzing custom code with Polyspace. The model contains a C Caller block that calls a function `counterbusFcn` defined in a file `hCounterBus.c` (declared in file `hCounterBus.h`). The model uses variables saved in a MAT file `dLctData.mat`, which is loaded in the model using a callback.



Copyright 2018 The MathWorks, Inc.

Run Analysis

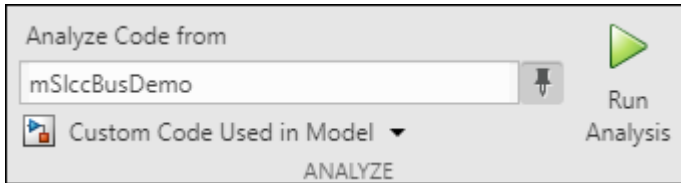
Configure analysis options and run Polyspace.

- 1 On the **Apps** tab, select **Polyspace Code Verifier** to open the **Polyspace** tab.
- 2 Specify the type of analysis:
 - Select the product to run, **Bug Finder** or **Code Prover**.

- Specify that the analysis must run on custom code in the model instead of generated code.

The **Analyze Code from** field shows the model name. Below the field, instead of **Code Generated as Top Model**, select **Custom Code Used in Model**.

- 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.

```
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.

Results List			
All results			
Family	File	Function	Status
-Run-time Check		1 37	
-Orange Check		1	
-Overflow		1	
?	hCounterBus.c	counterbusFcn()	Unreviewed
-Green Check		37	

The check highlights an addition operation in the `counterbusFcn` function that can overflow:

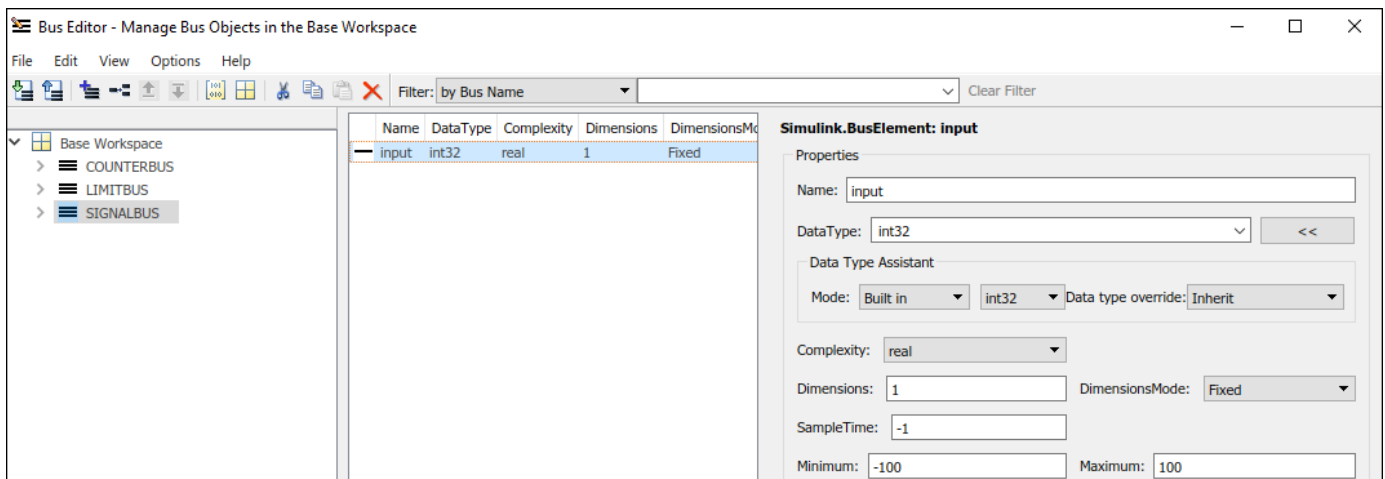
```
limit = u1->inputsignal.input + u2;
```

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

- The bus signal COUNTERBUS, which combines the signals input, upper_saturation_limit and lower_saturation_limit. The signal input is unbounded.
- The feedback from the C Caller block itself through a Delay block.

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

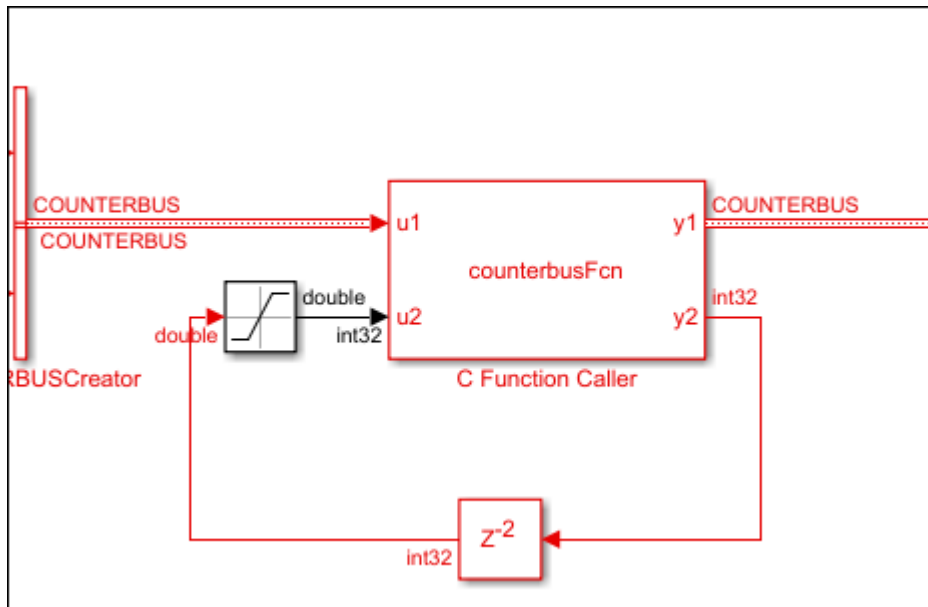
- 1 In the Simulink Editor, open the Model Explorer from the **Modeling** tab.
- 2 The base workspace variables contain the variable SIGNALBUS. Select this variable and open the bus editor to edit this variable. Specify a minimum and maximum value for the variable.



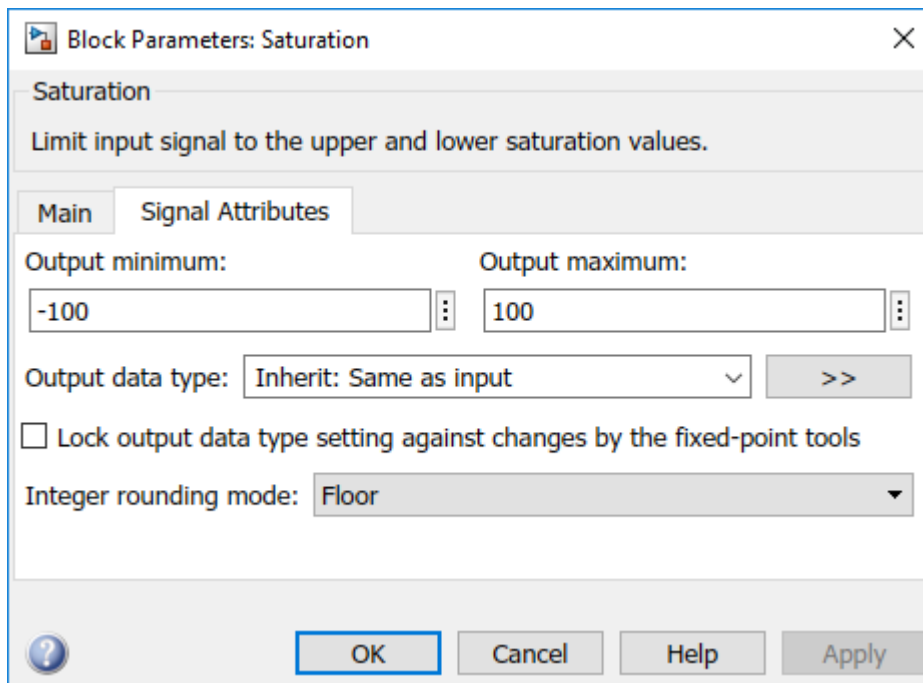
Save the bus object in a MAT-file. You can overwrite the file `dLctData.mat` or create a file.

You can also constrain the feedback from the C Caller block in several ways. For instance, you can saturate the feedback signal:

- 1 Add a Saturation block immediately before the feedback signal is input to the C Caller block.



- 2 On the **Signal Attributes** tab, specify a minimum and maximum value for the Saturation block output.



Note that specifying a lower and upper limit on the **Main** tab of the Saturation block is not sufficient to constrain the signal for the Polyspace analysis. The analysis uses the design ranges specified on the **Signal Attributes** tab.

Rerun the analysis. The **Overflow** check in the new set of results is green.

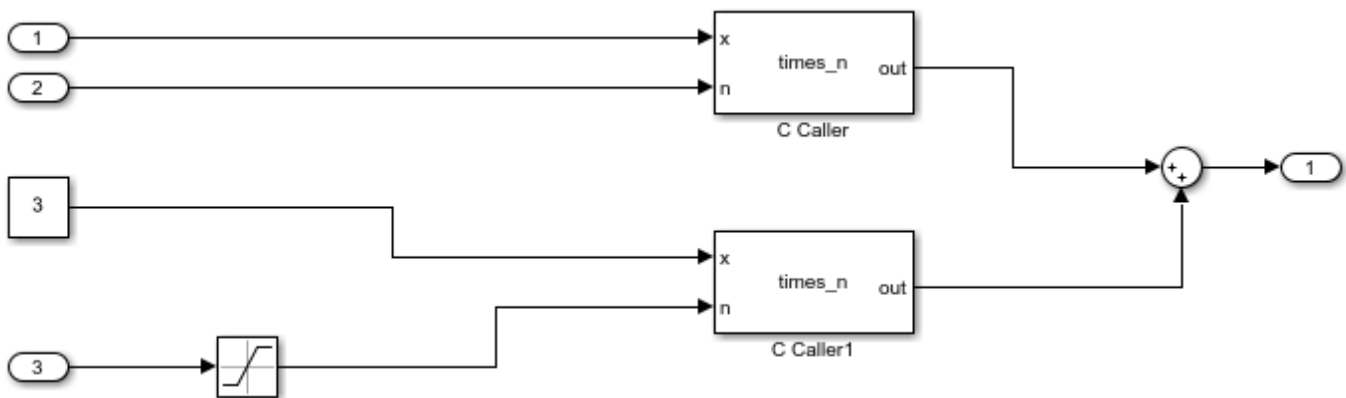
C/C++ Function Called Multiple Times in Model

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

The example runs a Code Prover analysis and shows how to determine the function call context starting from Code Prover results. Typically, in a Bug Finder analysis, you do not need to distinguish between different call contexts.

Open Model for Analyzing All Custom Code

Open the model `multiCcallerBlocks` for running Polyspace analysis.



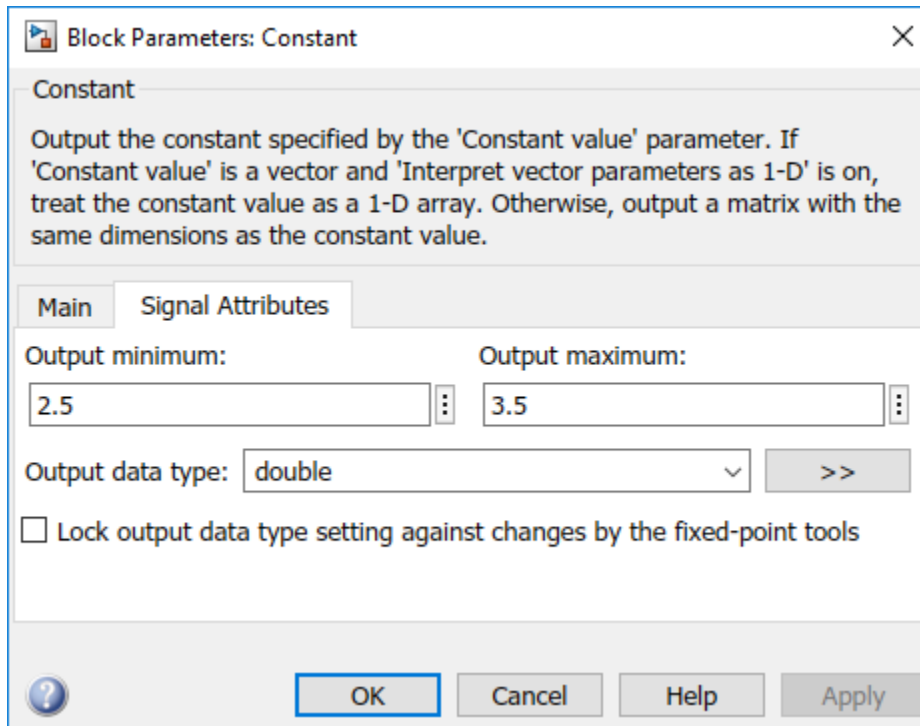
Inspect Model

The model contains two C Caller blocks calling the same function `times_n`. The inputs to one C Caller block come from two Inport blocks that have unbounded input. The inputs to the other C Caller block come from a Constant block and an Inport block that has the input bounded by a Saturation block.

To see the design ranges for the C Caller block that has bounded inputs:

- Double-click the Constant block or the Saturation block.
- On the **Signal Attributes** tab, note the design range.

For instance, although the Constant block has the constant value set to 3, the design range for verification is 2.5 to 3.5.



The design range for the **Saturation** block is [-1,1].

Run Analysis and Review Results

Run analysis as in the previous example and open the results.

The **Results List** pane shows an orange **Overflow** check. The product in the `times_n` function overflows.

```
#include "file.h"


double times_n(double x, double n) {
    return x * n;
}
```

Because the `times_n` function is called from two contexts, the orange color combines both contexts and might indicate two possible situations:

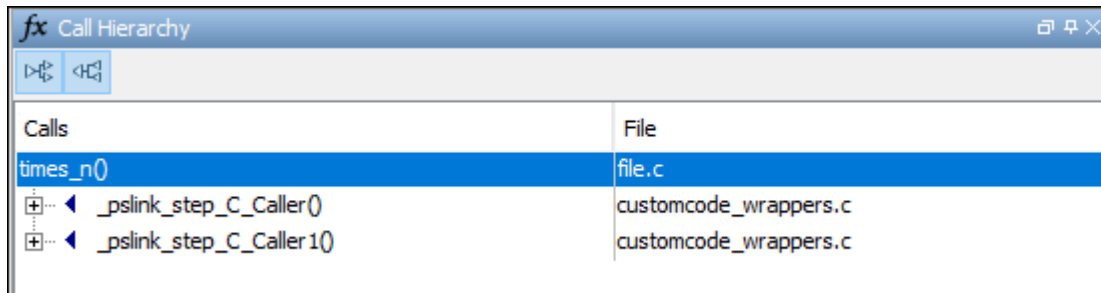
- The overflow occurs in both call contexts.
- The overflow is proven to not occur in one context (green check) and might occur in the other context (orange check).

To determine which call context leads to the overflow:

- 1 See all the callers of `times_n`.

Select the orange **Overflow** check. On the **Result Details** pane, click . The **Call Hierarchy** pane shows the callers of `times_n`.

- 2 On the **Call Hierarchy** pane, you see two wrapper functions as callers. Each wrapper function represents a C Caller block in the model.



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

Select one of the wrapper functions to open the source code for `customcode_wrappers.c`.

- 3 On the **Source** pane, inspect the code for the wrapper functions. To determine which inputs lead to the overflow, use tooltips on underlined inputs.

For instance, the wrapper function for the C Caller block that has bounded inputs looks similar to this code:

```
/* Go to model '<Root>/C Caller1' */
/* Variables corresponding to inputs for block C Caller1 */
real64_T _pslink_C_Caller1_In1;
real64_T _pslink_C_Caller1_In2;
/* Variables corresponding to outputs for block C Caller1 */
real64_T _pslink_C_Caller1_Out1;
/* Wrapper functions for code in block C Caller1 */
void _pslink_step_C_Caller1(void) {
    /* See tooltips on function inputs for input ranges */
    _pslink_C_Caller1_Out1 = times_n(_pslink_C_Caller1_In1, _pslink_C_Caller1_In2);
}
```

Use tooltips on the variables to determine their ranges. For instance, the tooltip on the variable `_pslink_C_Caller1_In1` shows that it is in the range [2.5, 3.5] and the tooltip on `_pslink_C_Caller1_In2` shows that it is in the range [-1,1]. Therefore, the product of the two inputs cannot overflow. The overflow must come from the other call context. You can see the tooltips on the inputs to the other call and confirm that the variables are unbounded.

To locate the C Caller block corresponding to a wrapper function, on the **Source** pane, click the blue block name link above the wrapper function (on the line starting with `Go to model`). The C Caller block is highlighted in the model.

Enable Context Sensitivity and Rerun Analysis


In this example, the function is simple enough that you can determine which call context leads to the overflow from the function inputs themselves. For more complex functions, you can configure the analysis to show results from the two contexts separately.

Because distinguishing call contexts involves a deeper analysis, the analysis might take longer. Therefore, enable context sensitivity only for select functions and only if you are not able to distinguish the call contexts by inspection.

In this example, to enable context sensitivity for the `times_n` function:

- 1 In your model, on the **Polyspace** tab, select **Settings > Project Settings**.

Alternatively, in the Polyspace user interface, select the **Project Browser**. Open the configuration of the project created for the analysis.

- 2 On the **Code Prover Verification > Precision** node, select `custom` for the option **Sensitivity context**. In the **Procedure** field, click  and enter `times_n`.

See also `Sensitivity context (-context-sensitivity)`.

Rerun the analysis from the model and reopen the results. Select the orange **Overflow** check.

The **Result Details** pane shows the call contexts separately. You can see that the overflow occurs only for the call with unbounded inputs (row with orange text) and does not occur for the other call (row with green text).

Click the row with orange text to directly navigate to the wrapper function leading to the orange check. From the wrapper function, you can navigate to the C Caller block with unbounded inputs.

? Overflow ?			
Warning: operation [*] on float may overflow (on MIN or MAX bounds of FLOAT64)			
Calling context	File	Scope	Line
operator * on type float 64 left: full-range [-1.7977E+308 .. 1.7977E+308] right: full-range [-1.7977E+308 .. 1.7977E+308]	customcode_wrappers.c	_pslink_step_C_Caller	26
operator * on type float 64 left: [2.5 .. 3.5] right: [-1.0 .. 1.0] result: [-3.5 .. 3.5]	customcode_wrappers.c	_pslink_step_C_Caller1	38

See Also

[pslinkoptions](#) | [pslinkrun](#)

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 5-2
- “Run Polyspace Analysis on S-Function Code” on page 5-21

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 design ranges and other context specific information specified in the Simulink model.

Prerequisites

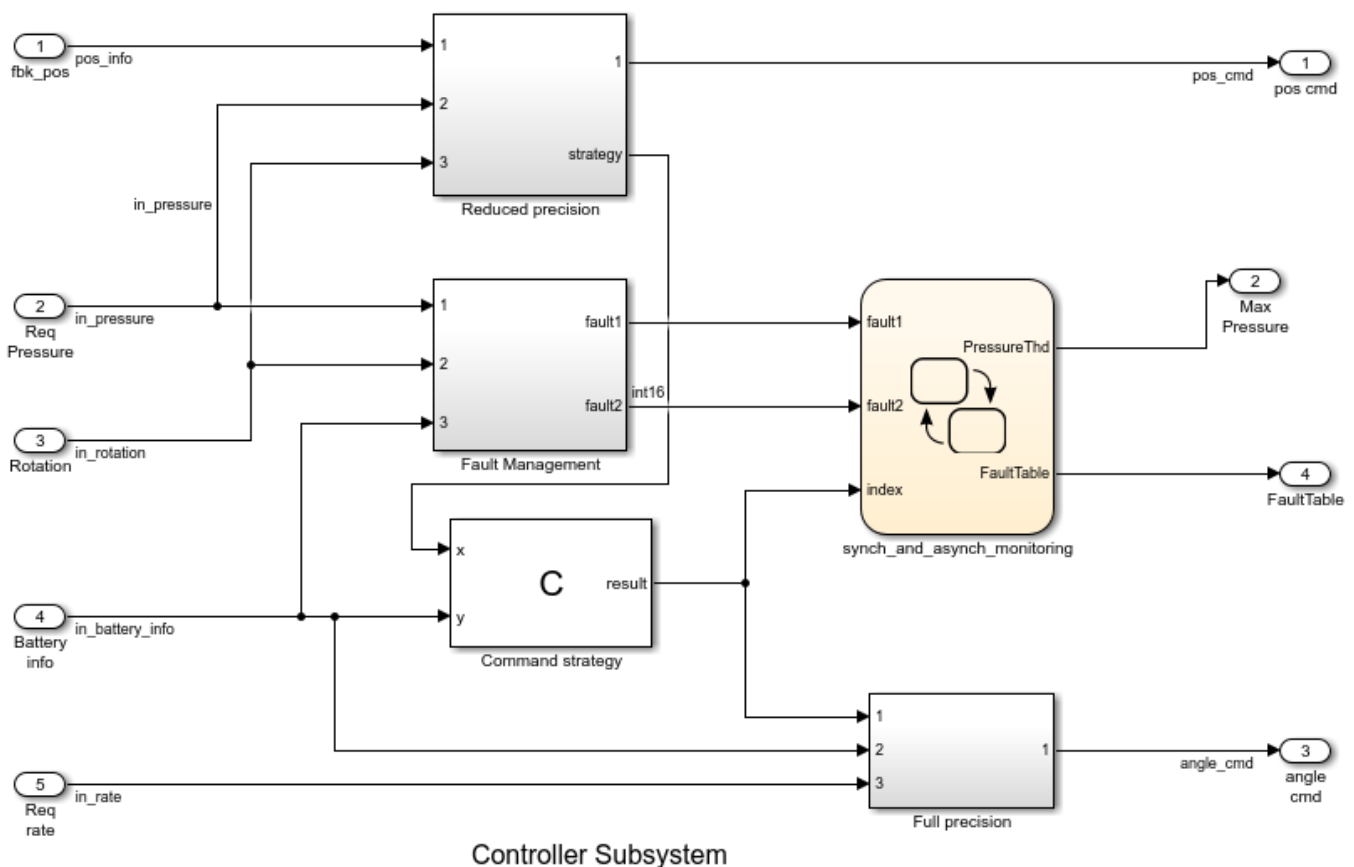
Before you run Polyspace from Simulink, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 3-2.

To open the model used in this example, search for this topic in the MATLAB Help browser and click the **Open Model** button. Alternatively, you can paste and run the following code from the MATLAB Command Window.

```
open_system('psdemo_model_link_sl_cscript');
```

Open Model for Running Polyspace Analysis on Custom Code in C Function Block

The model contains a C Function block called Command Strategy inside the controller subsystem.



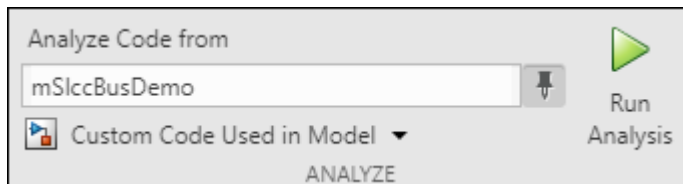
The **Command Strategy** block implements a look-up table using custom C code and outputs a value result based on two inputs *x* and *y*.


Run Polyspace Analysis

Run Polyspace Analysis from Simulink Editor

Click the **Apps** tab and select **Polyspace Code Verifier** to open the **Polyspace** tab.

- 1 Select **Bug Finder** or **Code Prover** from the drop-down list located at the leftmost corner of the **Polyspace** tab.
- 2 To run a Polyspace analysis on the custom C code in the C Function block, select **Custom Code Used in Model** from the drop-down list in the **Analyze** section.



- 3 To start the Polyspace analysis, click the **Run** button. The MATLAB Command Window displays the progress of the analysis.
- 4 After the analysis, to open the Polyspace user interface with the results, click the **Analysis Results** button. You can choose to open the results automatically after the analysis by selecting **Open results automatically after verification** in **Settings**.
- 5 To see all results of the Polyspace analysis, click **Clear active filters** from the **Showing** drop-down list in the **Results List** pane. If you run a **Code Prover** analysis, the results for the controller subsystem contain two red checks and an orange check.
- 6 To organize the results by family, click  and select **Family**.

Family	Information	File
Run-time Check		2 1 23
Red Check		2
Illegally dereferenced pointer		1
Out of bounds array index		1
Orange Check		1
Overflow		1
Green Check		23
Global Variable		
Not shared		

To switch between a **Bug Finder** and **Code Prover** analysis, return to the Simulink Editor from the Polyspace user interface. Select **Bug Finder** or **Code Prover** from the drop-down list located at the leftmost corner of the **Polyspace** tab and rerun the analysis.

Run Polyspace Analysis from MATLAB

You can run a Polyspace Code Prover analysis on custom code for this model from MATLAB Editor or the Command Window using this code:

```
% Load the model 'psdemo_model_link_sl_cscript'
load_system('psdemo_model_link_sl_cscript');
% Create a 'pslinkoptions' object
mlopts = pslinkoptions('psdemo_model_link_sl_cscript');
% Specify whether to run 'CodeProver' or 'BugFinder' Analysis
mlopts.VerificationMode = 'CodeProver';
% Specify custom code as analysis target and run the analysis
pslinkrun('-slcc', 'psdemo_model_link_sl_cscript', mlopts);
```

Identify Issues in C Code

To identify issues in the custom C code, use the information in the **Result Details** pane and the **Source** pane of the Polyspace user interface. If you do not see these panes, go to **Window > Show/Hide View** and select the missing pane. For details on the panes, see “Result Details” on page 16-30 and “Source” on page 16-24.

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 a full range 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 Polyspace Code Prover Results” on page 16-2.

Fix Identified Issues

Modify the custom C code or the model to fix the issues. You can fix a Polyspace check in several ways. The examples here illustrate the general workflow of fixing Polyspace checks.

Illegally dereferenced pointer

You can address this check in several ways. Modify the C code so that a nonexistent memory address is not accessed.

- 1 Return to the Simulink Editor and double-click on the C Function block to open the custom code.
- 2 Use the index operator on `array` to access a valid array index. You can access indices from 0 to 99 because `array` has 100 elements. Accessing indices beyond this range results in a run-time error in Simulink.

```
// access any index between 0 to 99
tmp = array[50] + 5;
```

Alternatively, assign the address of a valid memory location to `p` before the dereferencing operation. For example, `*p` can point to the 51st element in `array`.

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

Out of Bounds array index

You can address this check in several ways. Modify the code so that the size of `another_array[]` remains larger than or equal to the index value `return_val - i + 9`.

- 1 Return to the Simulink Editor and double-click on the C Function block to open the custom code.
- 2 Modify the prevailing condition on `return_val` so that the index value `return_val - i + 9` always evaluates to 0 or 1.

```
if ((return_val > 91) && (return_val < 92))
//...
```

Alternatively, declare `another_array` with size 19.

```
int another_array[19];
```

Overflow

You can address this check in several ways as well. Modify the C code or the model so that the range of the right side of the assignment operation remains equal to or larger than that of the left side.

- 1 Return to the Simulink Editor.
- 2 Saturate the input variables `x` and `y` in the model so that their difference can fit into a 16-bit integer. The workflow for fixing **Overflow** by using saturation blocks is described in “Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts” on page 5-23.

Alternatively, increase the size of `return_val` in the custom C code to accommodate `x - y`.

- 1 Return to the Simulink Editor and double-click on the C Function block to open the custom code.
- 2 Declare `return_val` as a 32-bit integer.

```
int32_T return_val;
```

For details about addressing Polyspace results, see “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

See Also

`pslinkoptions` | `pslinkrun`

More About

- “Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts” on page 5-23
- “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: <code>SystemTargetFile</code> (Simulink Coder) Value: An Embedded Coder Target Language Compiler (TLC) file. For example <code>ert.tlc</code> or <code>autosar.tlc</code> .	Location: Code Generation Name: System target file Value: Embedded Coder target file
	Name: <code>MatFileLogging</code> (Simulink Coder) Value: 'off'	Location: Code Generation > Interface Name: MAT-file logging Value: <input type="checkbox"/> Not selected
	Name: <code>GenerateReport</code> (Simulink Coder) Value: 'on'	Location: Code Generation > Report Name: Create code-generation report Value: <input checked="" type="checkbox"/> Selected
	Name: <code>IncludeHyperlinksInReport</code> (Embedded Coder) Value: 'on'	Location: Code Generation > Report Name: Code-to-model Value: <input checked="" type="checkbox"/> Selected
	Name: <code>GenerateSampleERTMain</code> (Embedded Coder) Value: 'off'	Location: Code Generation > Templates Name: Generate an example main program Value: <input type="checkbox"/> Not selected
	Name: <code>GenerateComments</code> (Simulink Coder) Value: 'on'	Location: Code Generation > Comments Name: Include comments Value: <input checked="" type="checkbox"/> Selected

Grouping	Command-Line	Name and Location in Configuration
Optimization	Name: DefaultParameterBehavior (Simulink Coder) Value: 'Inlined'	Location: Optimization Name: Default parameter behavior Value: Inlined
	Name: InitFltsAndDblsToZero (Simulink Coder) Value: 'on'	Location: Optimization Name: Use memset to initialize floats and doubles to 0.0 Value: <input type="checkbox"/> Not selected
	Name: ZeroExternalMemoryAtStartup (Embedded Coder) Value: 'off'	Location: Optimization Name: Remove root level I/O zero initialization Value: <input checked="" type="checkbox"/> Selected
Solver	Name: SolverType (Simulink) Value: 'Fixed-Step'	Location: Solver Name: Type Value: Fixed-step
	Name: Solver (Simulink) Value: 'FixedStepDiscrete'	Location: Solver Name: Solver Value: discrete (no continuous states)

Configure Advanced Polyspace Options in Simulink

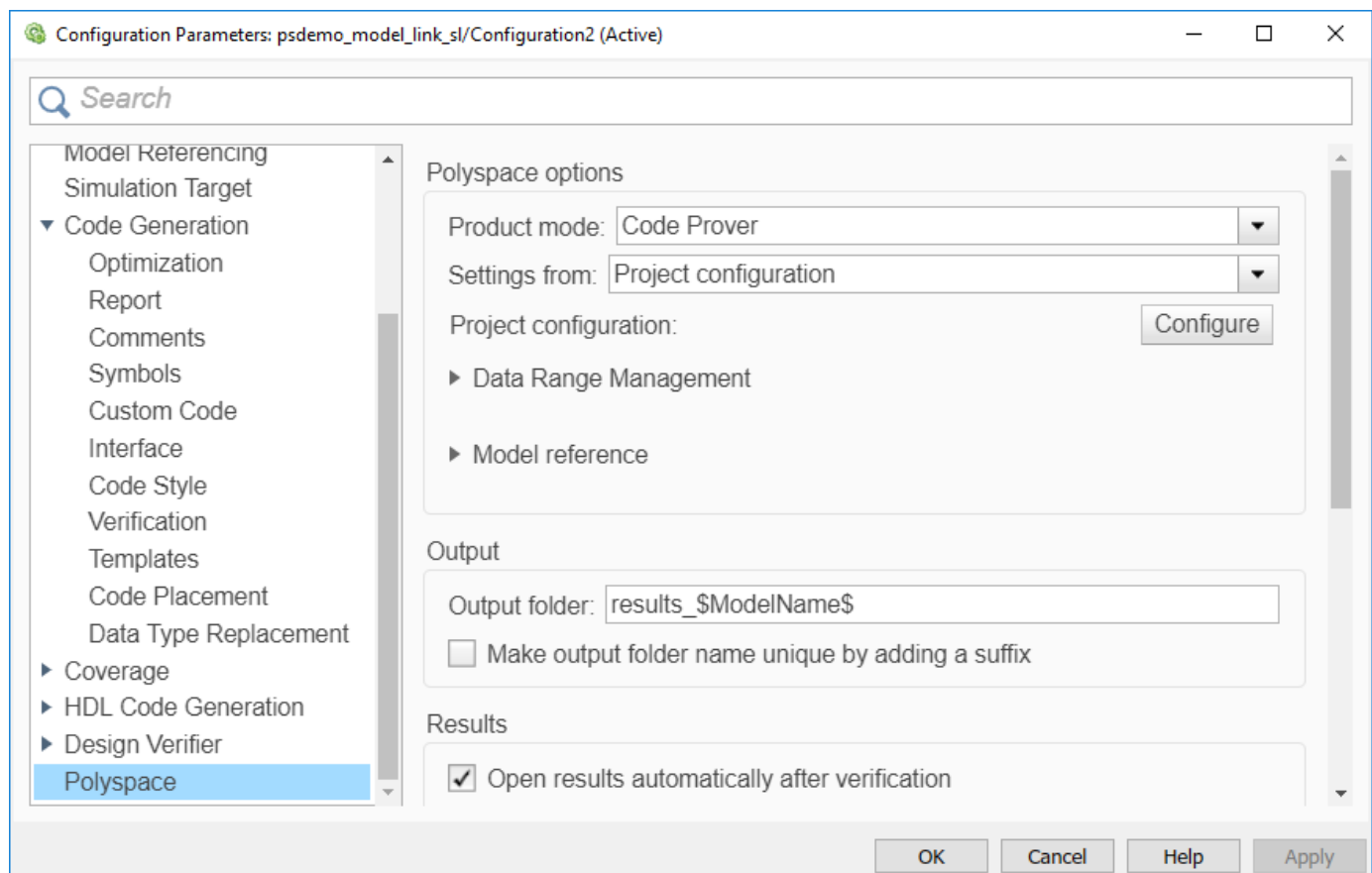
Before analyzing generated code in Simulink, you can change some of the default options. This topic shows how to configure the options and save this configuration.

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

Configure Options

Set basic options

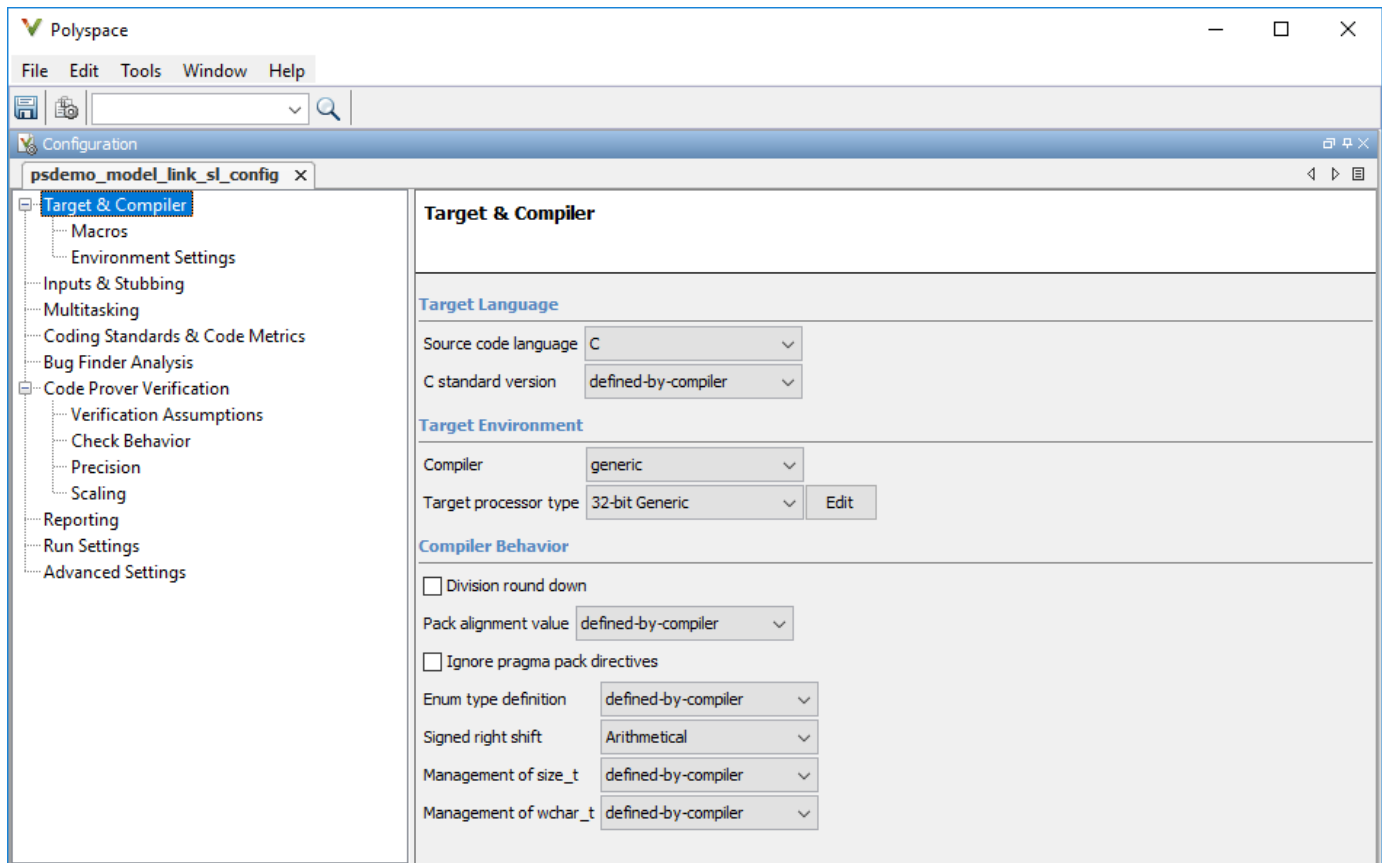
The commonly used options appear in Simulink Configuration Parameters.



To open the Polyspace options in the Simulink Configuration Parameters window, on the **Apps** tab, select **Polyspace Code Verifier**. Then, on the **Polyspace** tab, select **Settings** or **Settings > Polyspace Settings**.

Set advanced options

The more advanced options appear on the **Configuration** pane that also appears in the Polyspace user interface when you manually create a project for handwritten code.



To open the advanced options, on the **Polyspace** tab, select **Settings > Project Settings**.

On this pane, you can specify advanced settings such as:

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

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

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

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

- “Settings from (C)”: You select this basic option `Project configuration` and `MISRA C 2012 checking for generated code`.
- `Check MISRA C:2012 (-misra3)`: You disable this advanced option.

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


Share and Reuse Configuration

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

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

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

Set options from model

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

Reuse options in another model

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

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

If you want to check for additional issues, for instance MISRA C: 2012 violations, select `Project configuration` and `MISRA C 2012 checking for generated code`.

If you run an analysis from the command line, you can set these options with the `pslinkoptions` function. See also `pslinkoptions` Properties.

See Also

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 5-2
- “Run Polyspace Analysis on Code Generated with TargetLink” on page 5-49
- “Default Polyspace Options for Code Generated with Embedded Coder” on page 5-44
- “Default Polyspace Options for Code Generated with TargetLink” on page 5-52

How Polyspace Analysis of Generated Code Works

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

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

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

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

The `main` function conceptually looks like this:

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

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

For C++ code that is generated with Embedded Coder, the `initialize()`, `step()`, and `terminate()` functions and associated variables are either class members or have global scope.

Default Polyspace Options for Code Generated with Embedded Coder

In this section...
“Default Options” on page 5-44
“Constraint Specification” on page 5-44
“Recommended Polyspace options for Verifying Generated Code” on page 5-45
“Hardware Mapping Between Simulink and Polyspace” on page 5-45

Default Options

For Embedded Coder code, the software sets the following verification options by default:

```
-sources path_to_source_code
-D PST_ERRNO
-D main=main_rtwec
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
-functions-to-stub=[rtIsNaN,rtIsInf,rtIsNaNF,rtIsInfF]
-results-dir results
```

Note *matlabroot* is the MATLAB installation folder.

Constraint Specification

You can constrain inputs, parameters, and outputs to lie within specified ranges. Use these configuration parameters:

- “Input”
- “Tunable parameters”
- “Output”

The software automatically creates a Polyspace constraints file using information from the MATLAB workspace and block parameters.

You can also manually define a constraints file in the Polyspace user interface. See “Specify External Constraints” on page 10-2. If you define a constraints file, the software appends the automatically generated information to the constraints file you create. Manually defined constraint information overrides automatically generated information for all variables.

The software supports the automatic generation of constraint specifications for the following kinds of generated code:

- Code from standalone models
- Code from configured function prototypes
- Reusable code
- Code generated from referenced models and submodels

Additional Information

See also “External Constraints on Polyspace Analysis of Generated Code” on page 5-46.

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`

In addition, for the option `-server`, the software uses the value specified in the **Send to Polyspace server** check box on the **Polyspace** pane. These values override the corresponding option values in the **Configuration** pane of the Polyspace user interface.

You can specify other verification options for your Polyspace Project through the Polyspace **Configuration** pane. See “Configure Advanced Polyspace Options in Simulink” on page 5-39.

Hardware Mapping Between Simulink and Polyspace

The software automatically imports target word lengths and byte ordering (endianness) from Simulink model hardware configuration settings. The software maps **Device vendor** and **Device type** settings on the Simulink **Configuration Parameters > Hardware Implementation** pane to **Target processor type** settings on the Polyspace **Configuration** pane.

The software creates a generic target for the verification.

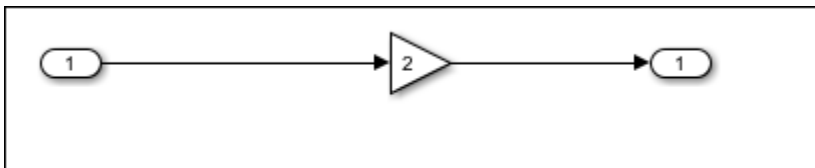
External Constraints on Polyspace Analysis of Generated Code

When you check generated code for bugs or run-time errors, you can choose whether to perform the check for all values of an input or a specific range of values. You can extract the input range from the Simulink model.

Likewise, you can use a fixed value for tunable parameters or a range of values. You can also check whether output values fall within a specific range.

Extract External Constraints from Model

Consider this simple model with an Inport block, a Gain block, and an Outport block. Suppose the signal in the Inport and Outport blocks and the gain parameter of the Gain block have a minimum and maximum value.



You can analyze the code generated from this model with these minimum and maximum values. On the **Apps** tab, select **Polyspace Code Verifier**. Then, on the **Polyspace** tab, select **Settings**. Specify these configuration parameters:

- “Input”: Select **Use specified minimum and maximum values**. The Code Prover analysis checks the generated code within the specified range of values from the Inport block. The Bug Finder analysis uses this information to exclude false positives.

Default: This option is selected.

- “Tunable parameters”: Select **Use specified minimum and maximum values**.

Default: This option is not selected. The analysis uses the fixed gain value of the Gain block (the value 2 in the example).

For the analysis to consider a range instead of a fixed value, the parameters must be tunable and not inlined. See **Default parameter behavior**.

- “Output”: Select **Verify outputs are within minimum and maximum values**. The Code Prover analysis creates a red check if the outputs exceed the range specified on the Outport block. See also **Correctness condition**.

Default: This option is not selected. The Code Prover analysis does not check output values.

After analysis, to check if a constrained range value is used, see one of these files:

- Constraint specification XML file `modelName_drs.xml` in the folder `results_modelname\modelname`.

- Polyspace project file `modelName.prpsj` in the folder `results_modelname`.

Open this file in the Polyspace user interface. In the project configuration, see the extracted constraints specified for the option `Constraint setup (-data-range-specifications)`.

Storage Classes Supported for Constraint Extraction

To allow constraint extraction from the Simulink model, the signals and parameters must have data types in specific storage classes. For details on storage classes, see “Choose Storage Class for Controlling Data Representation in Generated Code” (Embedded Coder).

Common Storage Classes

Storage Class	Signal Constraint Supported	Parameter Constraint Supported
Auto	Yes	Yes
ExportedGlobal	Yes	Yes
ImportedExtern	Yes	Yes
ImportedExternPointer	Yes	Yes
Model default	Yes	Yes

Other Storage Classes

Storage Class	Signal Constraint Supported	Parameter Constraint Supported
BitField	Yes	Yes
CompilerFlag	No	No
Const	No	Yes
ConstVolatile	No	Yes
Define	No	No
ExportToFile	Yes	Yes
FileScope	Yes	No
GetSet	No	No
ImportedDefine	No	No
ImportFromFile	No	No
Struct	No	No
Volatile	Yes	Yes

See Also

More About

- “Default Polyspace Options for Code Generated with Embedded Coder” on page 5-44
- “Choose Storage Class for Controlling Data Representation in Generated Code” (Embedded Coder)

Run Polyspace Analysis on Code Generated with TargetLink

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

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

Configure and Run Analysis

Configure code analysis

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

- Select the product to run: **Bug Finder** or **Code Prover**.
- Select **Settings**. Change default values of these options if needed.
 - “Settings from (C)”: Enable checking of MISRA or JSF[®] coding rules in addition to the default checks.
 - “Output folder”: Specify a dedicated folder for results. The default analysis runs Code Prover on generated code and saves the results in a folder `results_modelName` in the current working folder.
 - “Enable additional file list”: Add C files that are not part of the generated code.
 - “Open results automatically after verification”

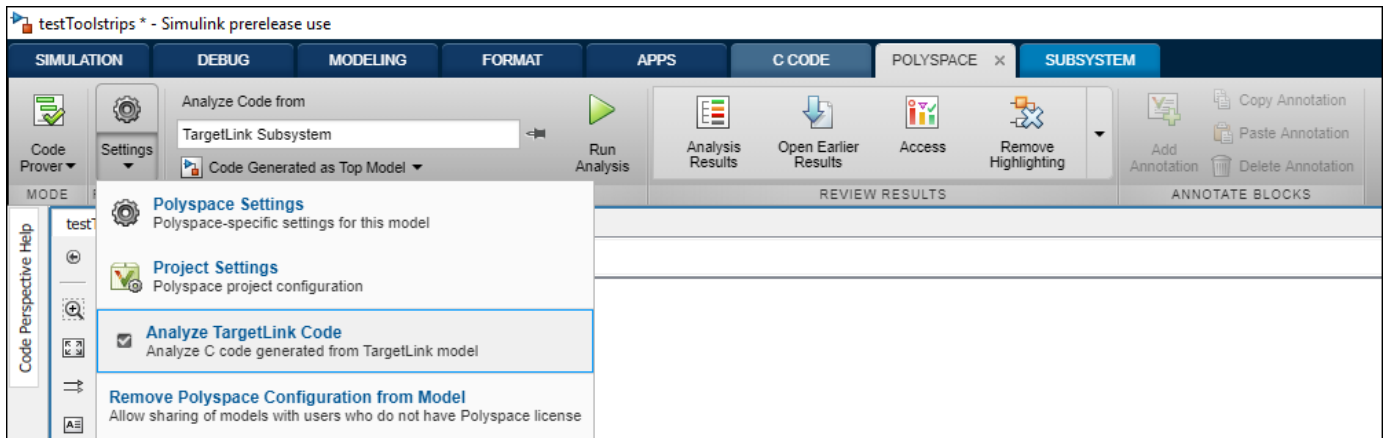
Analyze code

To analyze generated code:

- 1 Choose to analyze code generated from a TargetLink Subsystem. You cannot analyze code generated from the entire model.

The **Analyze Code from** field shows the top model. Unpin the content of this field and then select the TargetLink Subsystem.

- 2 Select **Settings > Analyze TargetLink Code**. Then, select **Run Analysis**.



You can follow the progress of the analysis in the MATLAB command window.

The results open automatically unless explicitly disabled. By default, the results are saved in a folder `results_ModelName` in the current folder. Each new run overwrites previous results. You can change these behaviors or save the results to a Simulink project using appropriate configuration parameters.

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).

Default Polyspace Options for Code Generated with TargetLink

In this section...

“TargetLink Support” on page 5-52
 “Default Options” on page 5-52
 “Lookup Tables” on page 5-52
 “Data Range Specification” on page 5-53
 “Code Generation Options” on page 5-53

TargetLink Support

The Windows version of Polyspace Code Prover is supported for versions 3.5 and 4.0 of the 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” on page 10-2. If you define a constraint file, the software appends the automatically generated information to the constraint file you create. Manually defined constraint information overrides automatically generated information for all variables.

Constraints cannot be applied to static variables. Therefore, the compilation flags `-D static=` is set automatically. It has the effect of removing the static keyword from the code. If you have a problem with name clashes in the global name space, either rename the variables or disable this option in Polyspace configuration.

Code Generation Options

From the TargetLink Main Dialog, it is recommended to:

- Set the option `Clean code`
- Unset the option `Enable sections/pragmas/inline/ISR/user attributes`
- Turn off the compute to overflow (CTO) generation. Polyspace can analyze code generated with CTO, but the results may not be as precise.

When you install Polyspace, the `tlcgOptions` variable is updated with `'PolyspaceSupport'`, `'on'` (see variable in `'C:\dSPACE\Matlab\Tl\config\codegen\tl_pre_codegen_hook.m'` file).

See Also

Related Examples

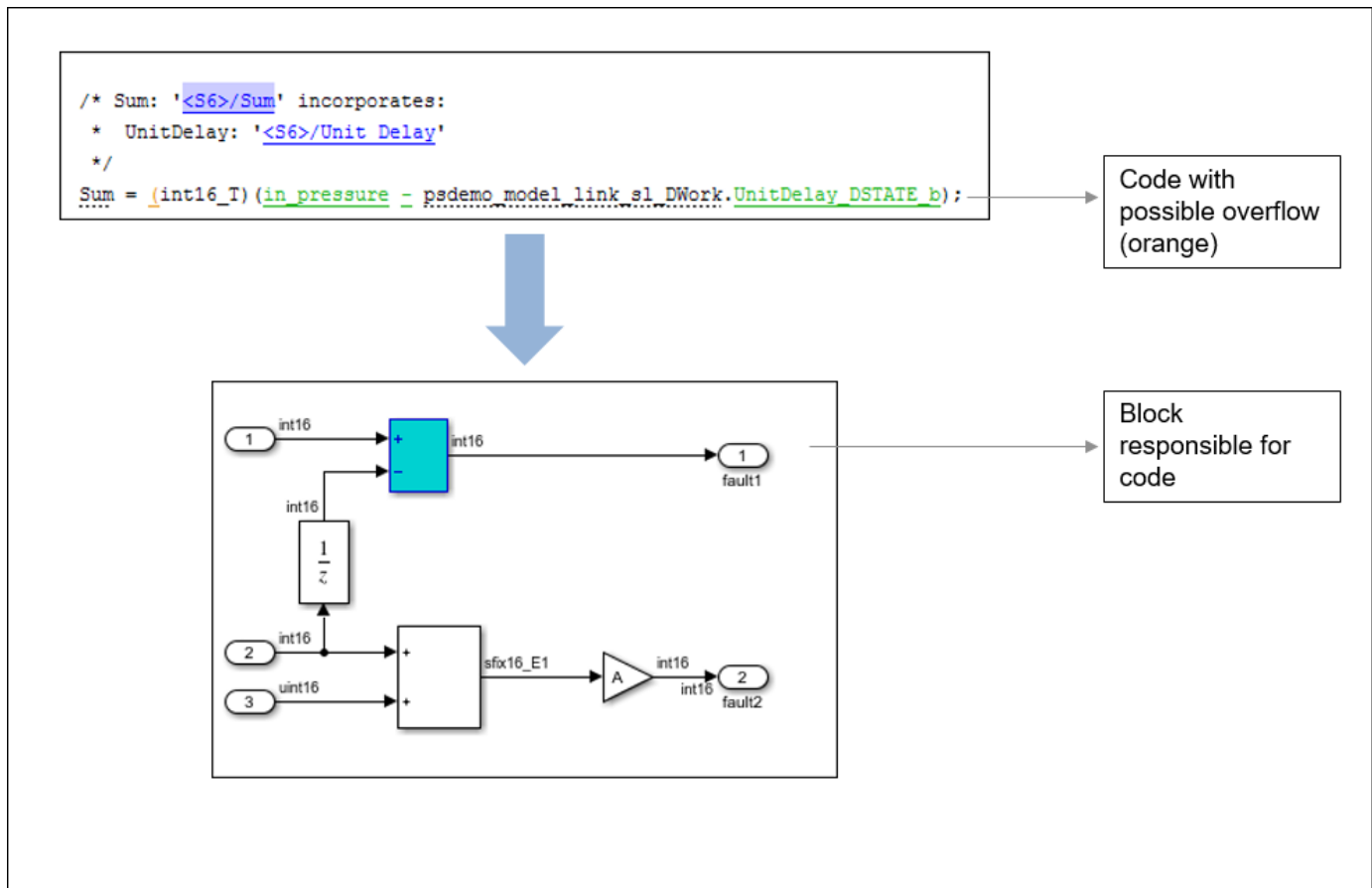
- “Run Polyspace Analysis on Code Generated with TargetLink” on page 5-49

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'

Run Polyspace on C/C++ Code Generated from MATLAB Code

After generating C/C++ code from MATLAB code, you can independently check the generated code for:

- Bugs or defects and coding rule violations: Use Polyspace Bug Finder.
- Run-time errors: Use Polyspace Code Prover.

Whether you generate code in the MATLAB Coder app or use `codegen`, you can follow the same workflow for checking the generated code.

This tutorial uses the MATLAB Coder example `averaging_filter` in `polyspaceroot\help\toolbox\codeprover\examples\matlab_coder`. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2020a`. The example shows a Code Prover analysis. You can follow a similar workflow for Bug Finder.

Prerequisites

To run this tutorial:

- You must have an Embedded Coder license. The MATLAB Coder app does not show options for running Polyspace unless you have an Embedded Coder license.
- You must be familiar with how to open and use the MATLAB Coder app or the `codegen` command. Otherwise, see the MATLAB Coder Getting Started.
- You must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” on page 3-2.

Run Polyspace Analysis

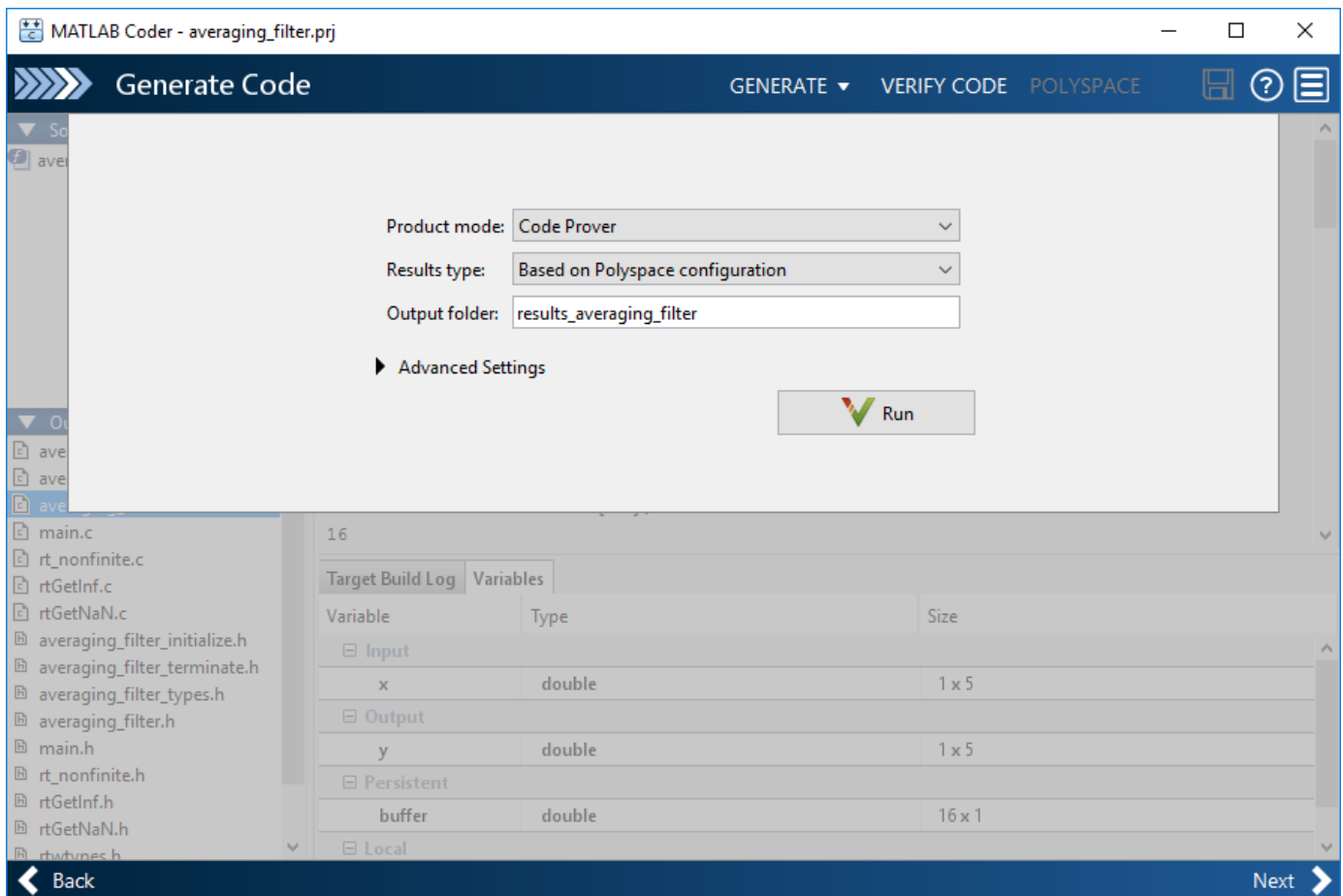
In the MATLAB Coder app, generate code from the file `averaging_filter.m` and analyze the generated code.

- 1 Generate code.

From the entry-point function in the file, generate standalone C/C++ code (a static library, dynamically linked library, or executable program) in the MATLAB Coder app. The function has one input. Explicitly specify a data type for the input, for instance, a 1 X 100 vector of type `double`, or provide a file for deriving data types.

- 2 Analyze the generated code.

After code generation, open the **Polyspace** pane and click **Run**.



If the analysis is completed without errors, the Polyspace results open automatically. If you close the results, you can reopen them from the final page in the app, under the section **Generated Output**. The results are stored in a subfolder `results_averaging_filter` in the folder containing the MATLAB file.

To script the preceding workflow, run:

```
% Generate code
matlabFileName = fullfile(polyspaceroot, 'help', ...
    'toolbox', 'codeprover', 'examples', 'matlab_coder', 'averaging_filter.m');
codegenFolder = fullfile(pwd, 'codegenFolder');
codegen(matlabFileName, '-config:lib', '-c', '-args', ...
    {zeros(1,100,'double')}}, '-d', codegenFolder);

% Configure Polyspace analysis
opts = pslinkoptions('ec');
opts.ResultDir = [tempdir 'results'];
opts.OpenProjectManager = 1;

% Run Polyspace
[polyspaceFolder, resultsFolder] = pslinkrun('-codegenfolder', codegenFolder, opts);
```

Review Analysis Results

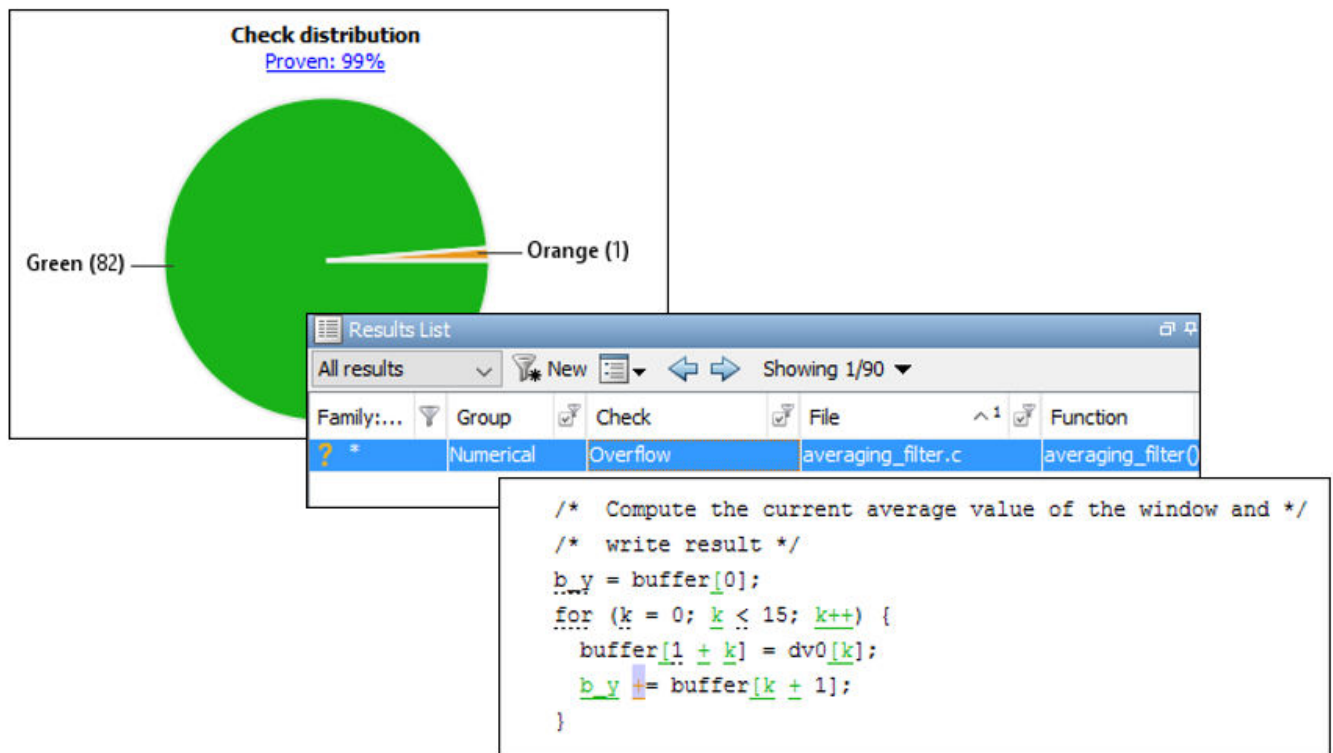
After analysis, the **Results List** pane shows a list of run-time checks. For an explanation of the result colors, see “Code Prover Result and Source Code Colors” on page 16-8.

Review the results and determine whether to fix the issues.

- 1 Filter out results that you do not want to review. For instance, you might not want to see the green checks.

See an overview of the results on the **Dashboard** pane. Click the orange section of the pie chart to filter the list of results on the **Results List** pane to the one orange check. Click this orange **Overflow** check and see the source code for the operation that can overflow.

If results are grouped by family, to see a flat list, on the **Results List** pane, from the  dropdown, select **None**.



Check distribution
Proven: 99%

Green (82) — Orange (1)

Family:...	Group	Check	File	Function
?	Numerical	Overflow	averaging_filter.c	averaging_filter()

```

/* Compute the current average value of the window and */
/* write result */
b_y = buffer[0];
for (k = 0; k < 15; k++) {
    buffer[1 + k] = dv0[k];
    b_y += buffer[k + 1];
}

```

- 2 Find the root cause of each run-time error.

On the **Source** pane, use right-click navigation tools and tooltips to identify the root cause of the check. In this case, you see that the + operation overflows because Polyspace makes an assumption about the input array to the function. The assumption is that the array elements can have any value allowed by their `double` data type. The tooltip on the line `buffer[0] = x[i]` shows the assumed range.

```

/* Add a new sample value to the buffer */
buffer[0] = x[i];

/* Com Assignment to element of static array (float 64): [-1.7977E+308 .. 1.7977E+308]
/* wri
b_y = b array size: 16
for (k array index value: 0
  buffe
  b_y += buffer[k + 1];
}

```

With an Embedded Coder license, you can easily trace back from the generated C code to the original MATLAB code. See “Interactively Trace Between MATLAB Code and Generated C/C++ Code” (Embedded Coder).

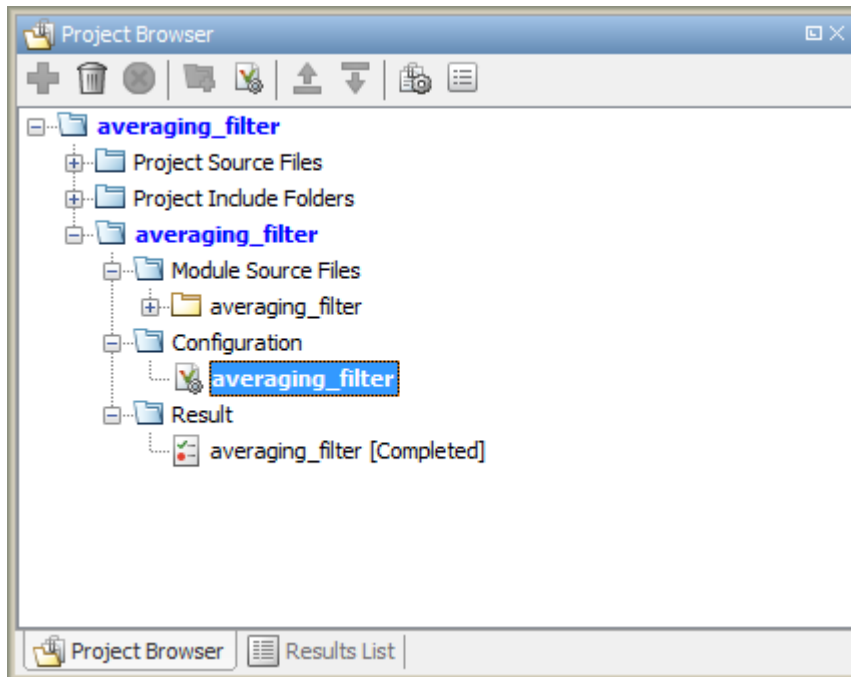
Run Analysis for Specific Design Range

You can check the generated code for a specific range of inputs. Range specification helps narrow down the default assumption that inputs are full-range.

To specify a range for inputs:

- 1 Open the analysis configuration.

In the Polyspace user interface, switch to the Polyspace project created for the analysis. Select **Window > Reset Layout > Project Setup**. On the **Project Browser** pane, click the project configuration.



- Specify a design range for the inputs.

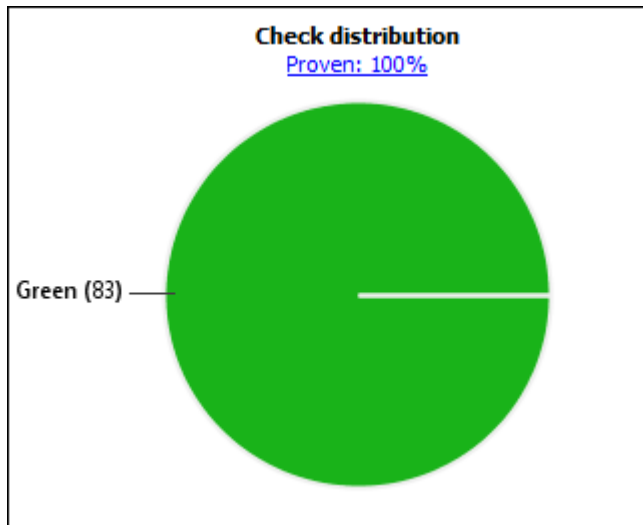
In the **Configuration** pane, on the **Inputs & Stubbing** node, set up your constraints. Click **Edit** beside **Constraint setup**. Constrain the range of the first input to [-100..100].

Name	File	Main Generator Called	Init Mode	Init Range
Global Variables				
User Defined Functions				
averaging_filter()	averaging_filter.c	MAIN GENERATOR		
averaging_filter.arg1	averaging_filter.c		INIT	
averaging_filter.* arg1	averaging_filter.c		INIT	-100..100
averaging_filter.arg2	averaging_filter.c		INIT	

You can overwrite the default constraint template or save the constraints elsewhere. For information on the columns in this window, see “External Constraints for Polyspace Analysis” on page 10-7.

- Rerun the analysis from the Coder app (or at the MATLAB command line) and see the results.

On the **Dashboard** pane, you do not see the previous orange overflow anymore.



See Also

pslinkrun

More About

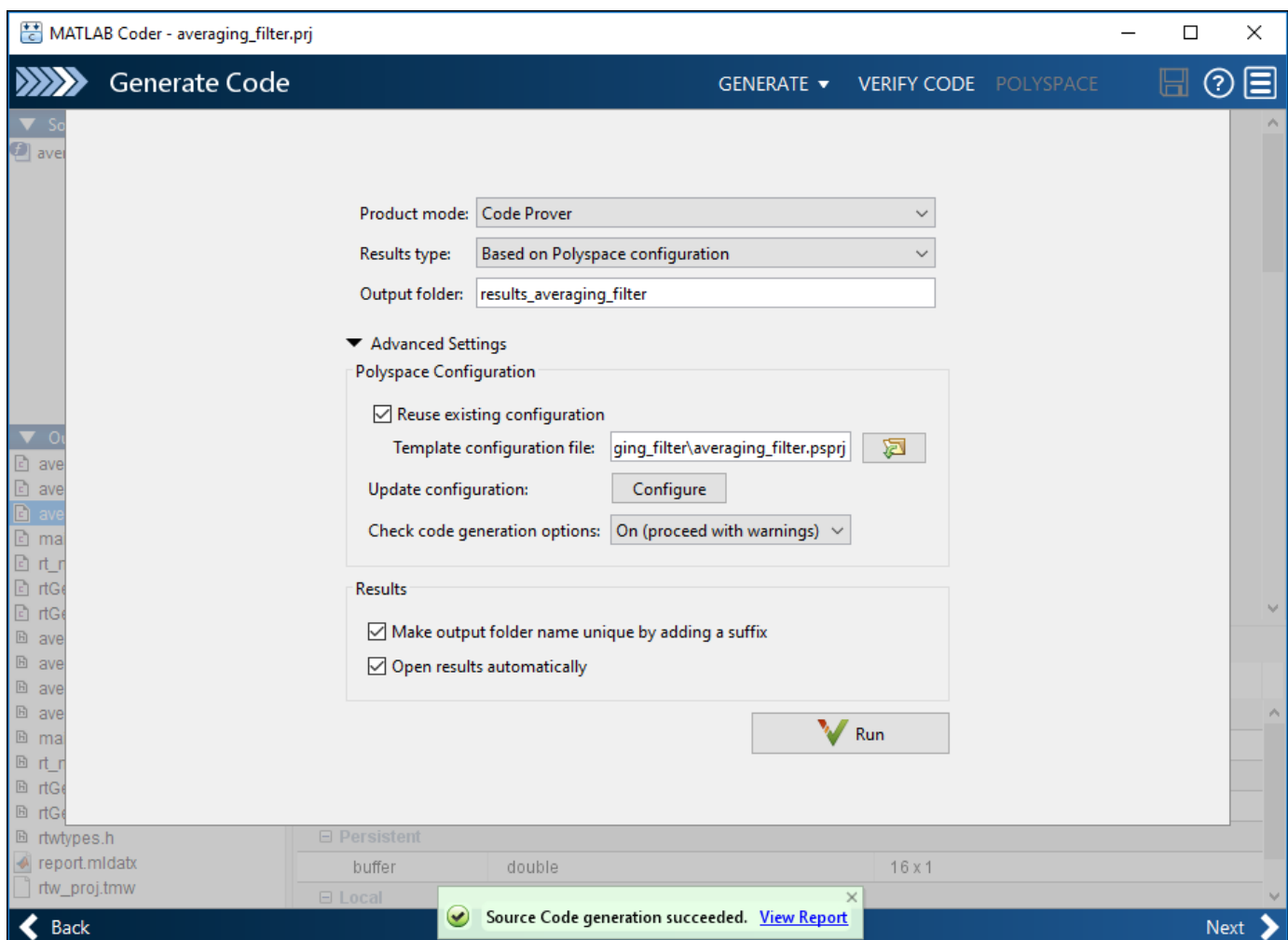
- "Configure Advanced Polyspace Options in MATLAB Coder App" on page 5-62

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 5-56.

Configure Options



The default analysis runs Code Prover based on a default project configuration. The results are stored in a folder `result_project_name` in the current working folder.

You can change these options in the MATLAB Coder App itself:

- **Product mode:** Select Code Prover or Bug Finder.
- **Results type:** Check for MISRA C:2004 (MISRA AC AGC) or MISRA C:2012 rule violations, in addition to or instead of the default checkers.
- **Output folder:** Choose an output folder name. To save the results of each run in a new folder, under **Advanced Settings**, select **Make output folder name unique by adding a suffix**.
- **Check code generation options:** Choose to see warnings or errors if the code generation uses options that can result in imprecise Code Prover analysis.

For instance, if the code generation setting **Use memset to initialize floats and doubles to 0.0** is disabled, Code Prover can show imprecise orange checks because of approximations. See “Orange Checks in Code Prover” on page 16-48.

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 (Polyspace Bug Finder) 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

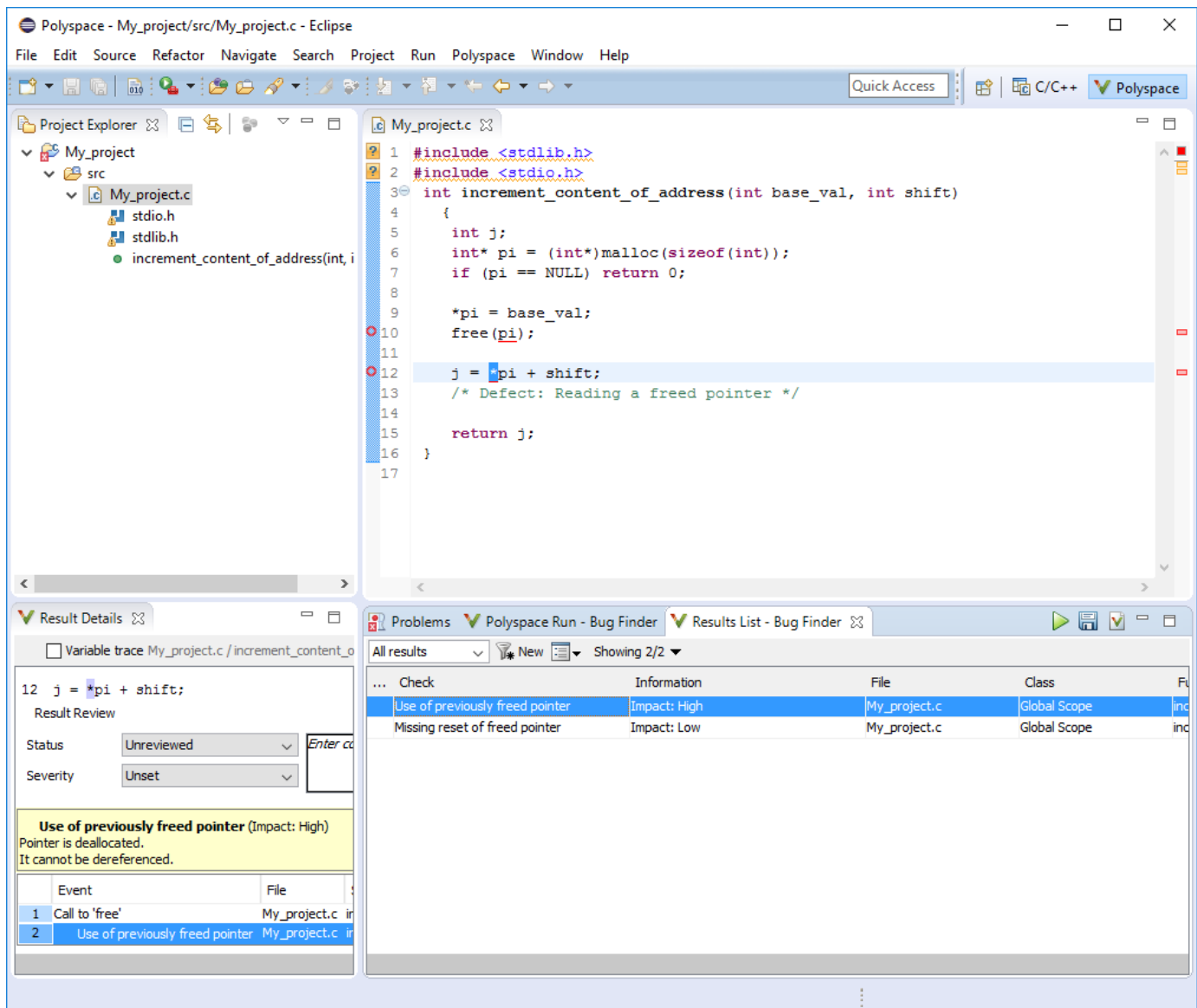
- “Run Polyspace on C/C++ Code Generated from MATLAB Code” on page 5-56

Run Polyspace Analysis in IDE Plugins

Run Polyspace Analysis in Eclipse

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.

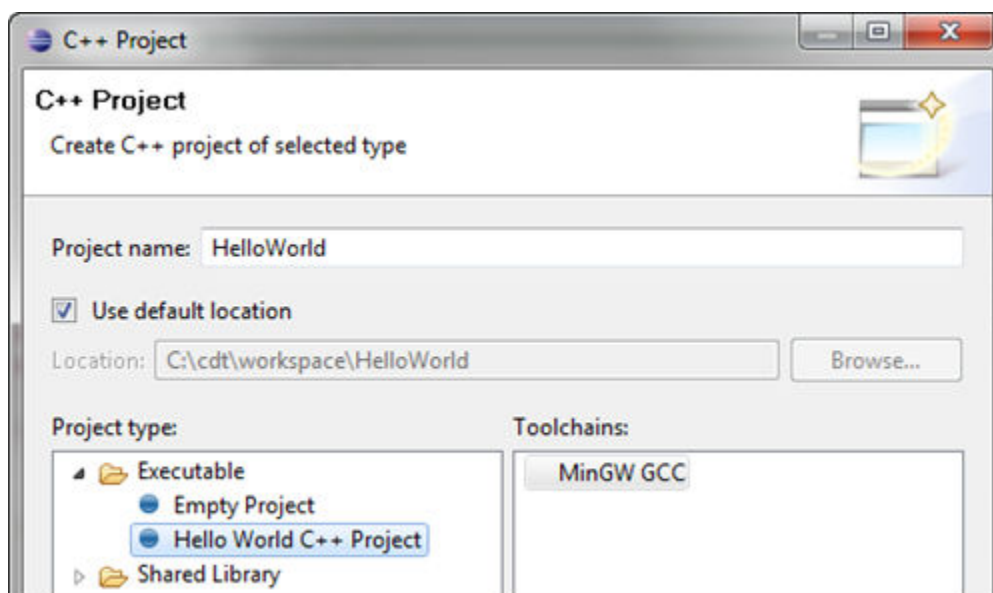
The analysis progress bar, quick run buttons and analysis results appear on specific panes. To avoid cluttering your window, you can confine these panes to the Polyspace perspective. Select **Window > Open Perspective > Other**. In the Open Perspective dialog box, select **Polyspace**. You can switch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

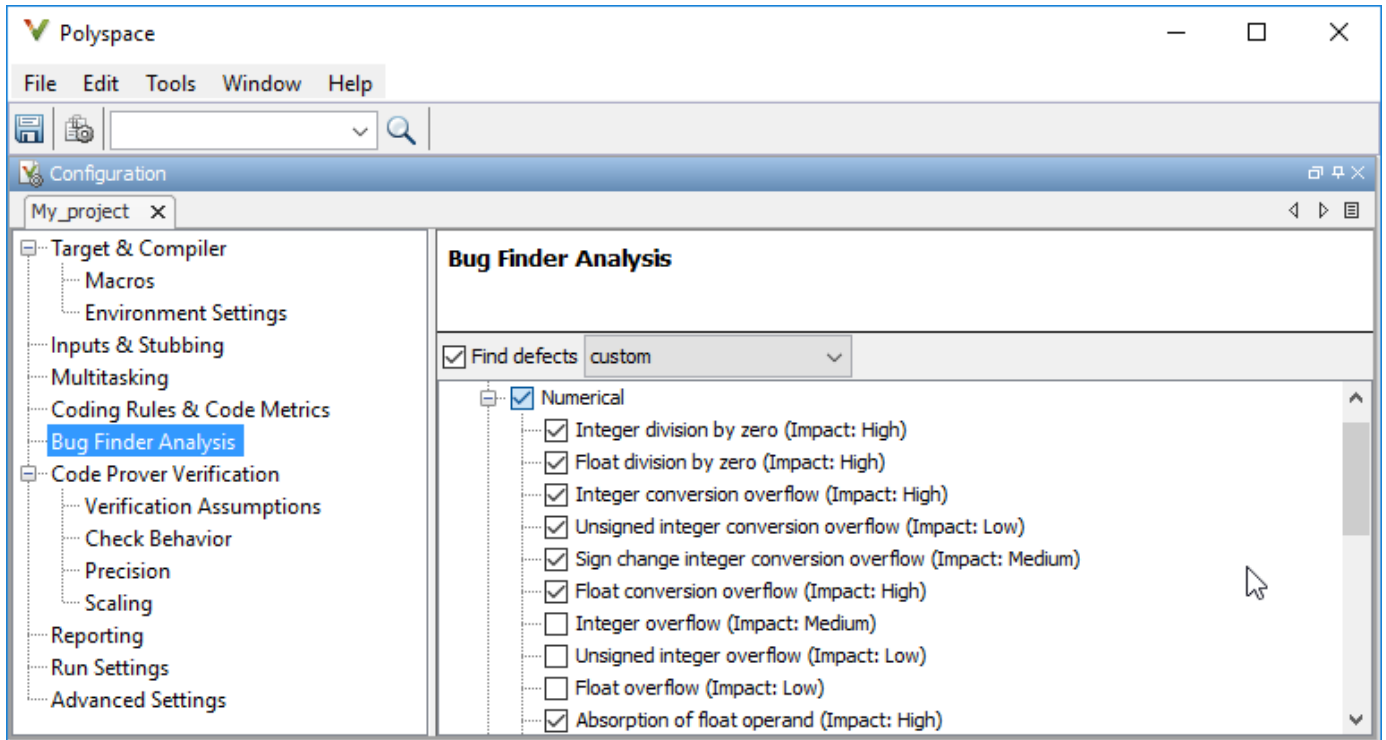


If your Eclipse project uses a build command (makefile) that has the compiler information, you must perform some additional steps to extract this information for the Polyspace analysis.

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 6-7.

- **Other analysis options:** You can retain the default analysis options or adjust them to your requirements. Select **Polyspace > Configure Project**.




The key options are:

- **Target & Compiler:** If you have not specified your compiler information through your Eclipse project, use these options.
- **Bug Finder Analysis:** Specify which defects to check for in a Bug Finder analysis.
- **Code Prover Verification, Check Behavior, Precision:** Modify the behavior of checkers in a Code Prover verification.

Note that you cannot run a remote analysis using the Polyspace plugin for Eclipse. To send the analysis job to a remote cluster, start the analysis from the Polyspace user interface or using scripts. See “Polyspace Analysis on Clusters”.

Run analysis

After configuration, you can start and stop a Polyspace analysis explicitly from the **Polyspace** menu, right-click options on your Eclipse project or quick run buttons in the Polyspace panes. You can switch between Bug Finder and Code Prover using the  icon on the **Polyspace Run** pane.

Run analysis when saving code

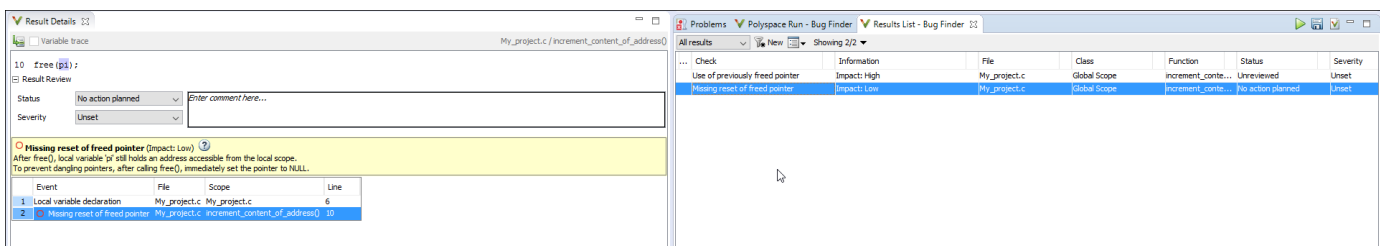
In the Polyspace perspective, you can set up a Bug Finder analysis that runs each time you save your code. To enable this analysis, select **Polyspace > Run Fast Analysis on Save**. The analysis runs

quickly but looks for a reduced set of defects. You get the same results as if you had specified the analysis option Use fast analysis mode for Bug Finder (-fast-analysis).



Review Analysis Results

View results after analysis

After analysis, the results appear on the **Results List** pane. Click each result to see the source code and details on the **Result Details** pane.



View results as available

Some results of a Bug Finder analysis are often available before the analysis is complete. If so, the  icon in the **Polyspace Run - Bug Finder** pane turns to . To load available results, click this icon. The icon shows up again when more results are available.

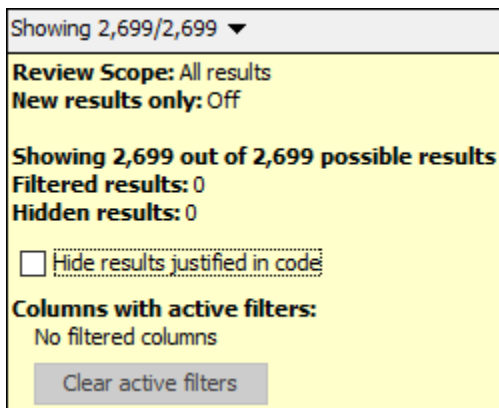
Address results

Based on the result details, fix your code or justify the result. To justify a result, set its **Status** to **Justified**, **No Action Planned** or **Not a Defect**. To hide a justified result in the next run, add the status as annotation to your source code. See “Annotate Code and Hide Known or Acceptable Results” on page 18-6.

For quick annotation, right-click the result and select **Annotate Code and Hide Result**. The option adds annotations in this format and hides the result from the results list:

```
line of code; /* polyspace Family:Result_name */
```

For details of the format, see “Annotate Code and Hide Known or Acceptable Results” on page 18-6. To unhide the hidden results, from the **Showing** menu, clear the box **Hide results justified in code**.



See Also

Related Examples

- “Specify Polyspace Compiler Options Through Eclipse Project” on page 6-7
- “Interpret Polyspace Code Prover Results” on page 16-2
- “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2
- “Filter and Group Results” on page 19-2

Specify Polyspace Compiler Options Through Eclipse Project

Polyspace analysis in Eclipse uses a set of default analysis options preconfigured for your coding language and operating system. For each project, you can customize the analysis options further.

- **Compiler options:** You specify the compiler that you use, the libraries that you include and the macros that are defined for your compilation.
- If your Eclipse project directly refers to a compilation toolchain, the analysis extracts the compiler options from the project.

See “Eclipse Refers Directly to Your Compilation Toolchain” on page 6-7.

- If the project refers to your compilation toolchain through a build command, the analysis cannot extract the compiler options. Trace the build command to extract the options.

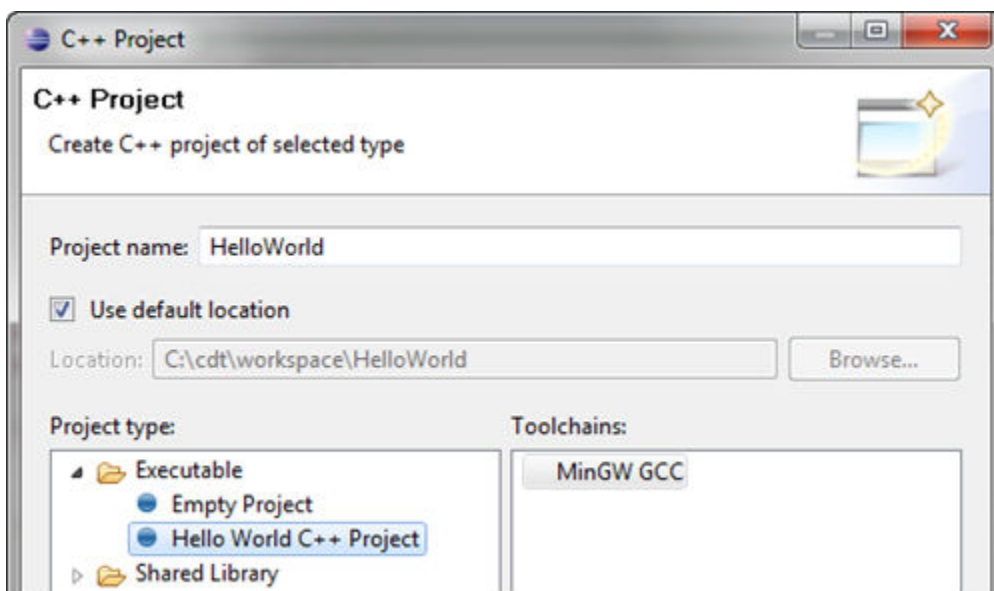
See “Eclipse Uses Your Compilation Toolchain Through Build Command” on page 6-8.

- **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 in Eclipse” on page 6-2.

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.

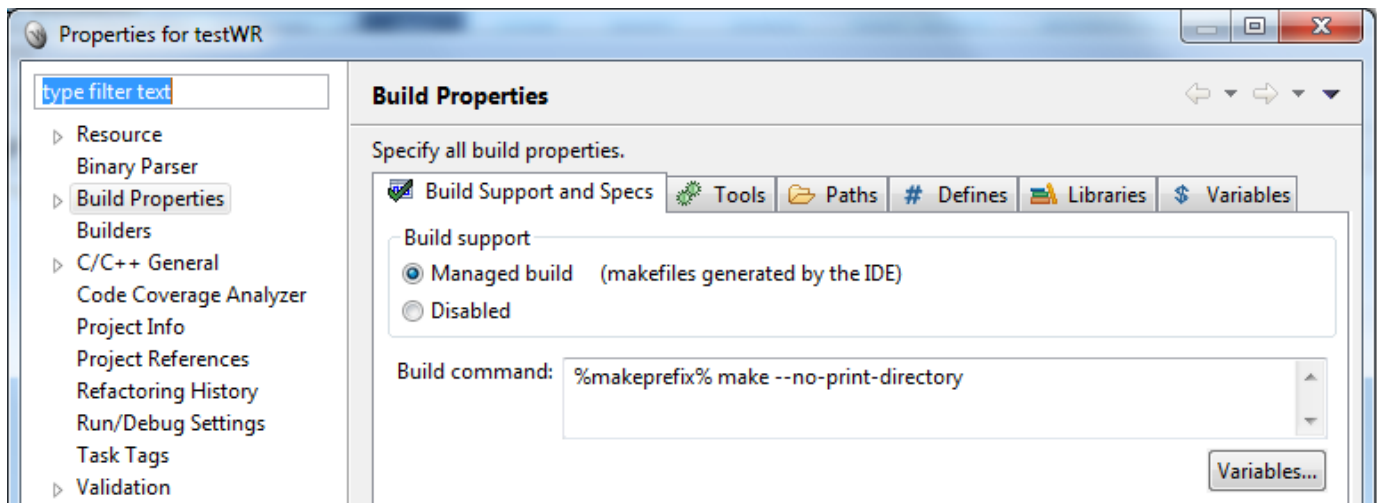
During analysis, the paths are implicitly used with the analysis option `Include folders (-I)`.

- 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\Projects\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 in Eclipse” on page 6-2

Running Polyspace on AUTOSAR Code

- “Using Polyspace in AUTOSAR Software Development” on page 7-2
- “Benefits of Polyspace for AUTOSAR” on page 7-5
- “Run Polyspace on AUTOSAR Code” on page 7-12
- “Troubleshoot Polyspace Analysis of AUTOSAR Code” on page 7-17
- “Run Polyspace on AUTOSAR Code with Conservative Assumptions” on page 7-21
- “Run Polyspace on AUTOSAR Code Using Build Command” on page 7-23

Using Polyspace in AUTOSAR Software Development

Whatever your role in the AUTOSAR software development workflow, you can benefit from Polyspace. These sections describe some of the situations where you can use Polyspace to check the C code implementation of software components.

For an overview of Polyspace for AUTOSAR, see “Benefits of Polyspace for AUTOSAR” on page 7-5.

Check if Implementation of Software Components Follow Specifications

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 Metrics.
- 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 Metrics after verification. Otherwise, you can upload them later.

See “Run Polyspace on AUTOSAR Code” on page 7-12.

- 2 Upload all results to Polyspace Metrics. When uploading, make sure you use the same project name and version number for all results.

See “Generate Code Quality Metrics” on page 21-9.

- 3 In Polyspace Metrics, click the project name and see a summary of the results.

Verification	Verification Status	Run-Time Errors						Review Progress
		Confirmed Defects	Run-Time Selectivity	Green	Red	Orange	Gray	
1.0	completed (PASS2)		98.6%	338	1	5	1	0.0%
pkg.tst002.swc001.bhv001	completed (PASS2)		98.4%	308	1	5		0.0%
pkg.tst002.swc002.bhv	completed (PASS2)		100.0%	30			1	0.0%

See “View Code Quality Metrics” on page 21-12.

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” on page 20-2.

Assess Impact of Edits to Specifications

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 7-12.
- 2 Upload the result for a software component to Polyspace Metrics.
See “Generate Code Quality Metrics” on page 21-9.
- 3 Rerun Polyspace using the updated specifications.
- 4 Upload the new result to Polyspace Metrics. Use the same project name but a different version number when uploading the result.
- 5 See if there is an increase in the number of red, gray or orange checks.

See “View Trends in Code Quality Metrics” on page 21-20.

Check Code Implementation for Run-time Errors and Mismatch with Specifications

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 7-12.

Check Code Implementation Against Specification Updates

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 7-5
- “Run Polyspace on AUTOSAR Code” on page 7-12
- “Review Polyspace Results on AUTOSAR Code” on page 17-78

Benefits of Polyspace for AUTOSAR

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.

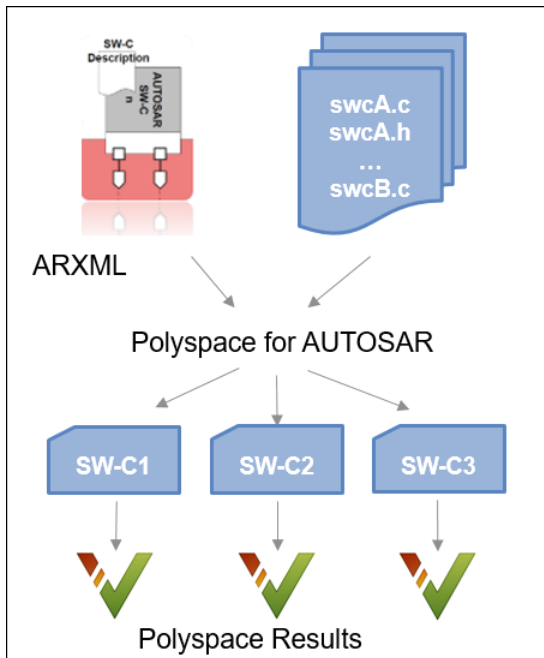
- Run-time errors: These checks aim to prove the absence of certain types of run-time errors in the bodies of the runnables (for instance, overflow). The proof uses the specifications in the ARXML to determine precise ranges for runnable arguments and RTE function return values and output arguments. 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 7-12
- “Review Polyspace Results on AUTOSAR Code” on page 17-78

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>
        ...
    </SWC-INTERNAL-BEHAVIOR>
  </INTERNAL-BEHAVIORS>
</APPLICATION-SW-COMPONENT-TYPE>
```

```

</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.

```

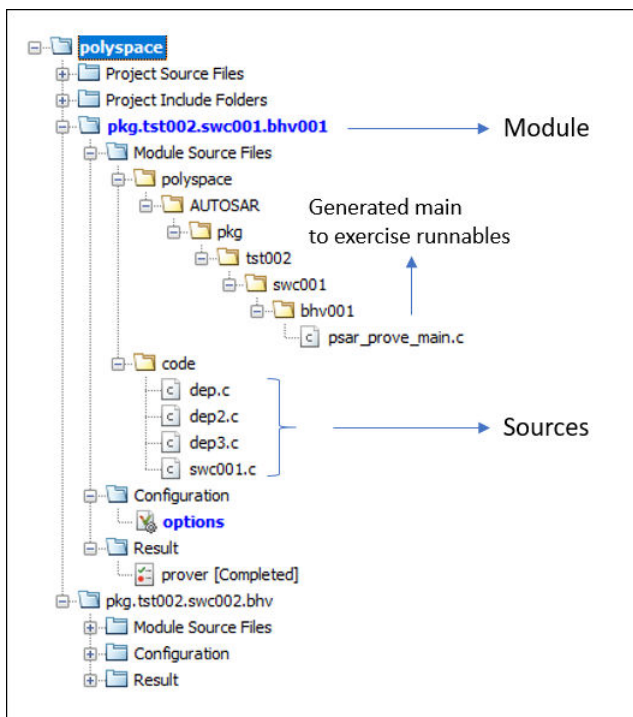
iOperations_ApplicationError foo(
    Rte_Instance const self,
    app_Array_2_n320to320ConstRef aInput,
    app_Array_2_n320to320Ref aOutput,
    app_Enum001Ref aOut2)
{
    /* Your implementation */
}

```

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);
```

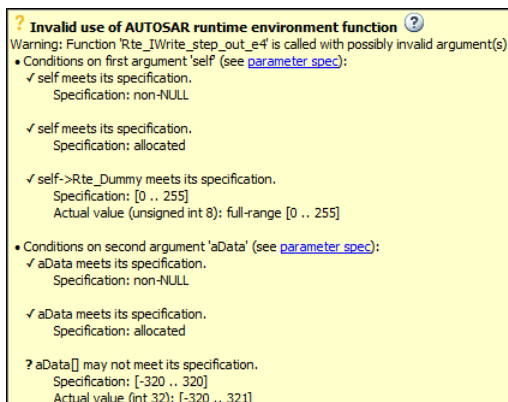
the second argument of `Rte_IWrite_step_out_e4` can have the previously defined data type `n320to320`. But at run time, your code implementation can pass a value outside the range `[-320, 320]`. The argument might be the result of a series of previous operations and one of those operations can cause 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 16-42). 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 7-2
- “Run Polyspace on AUTOSAR Code” on page 7-12
- “Review Polyspace Results on AUTOSAR Code” on page 17-78

Run Polyspace on AUTOSAR Code

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 3.0 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.

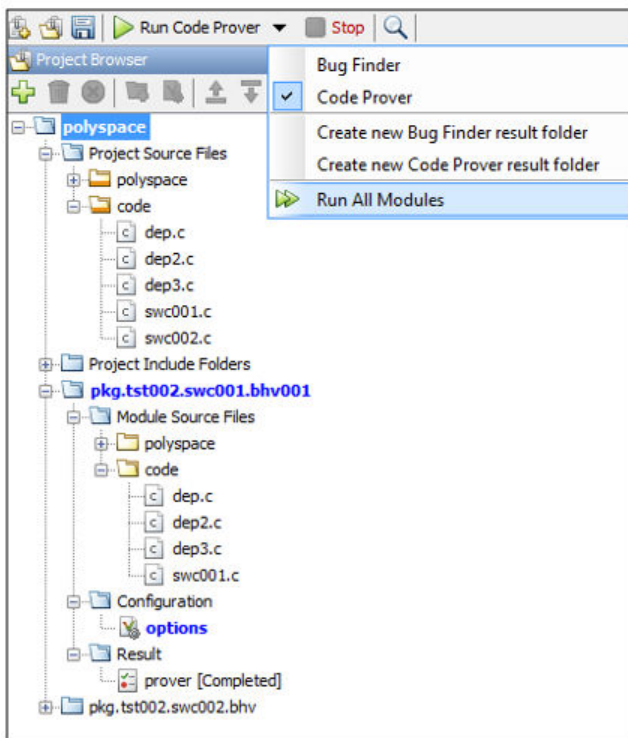
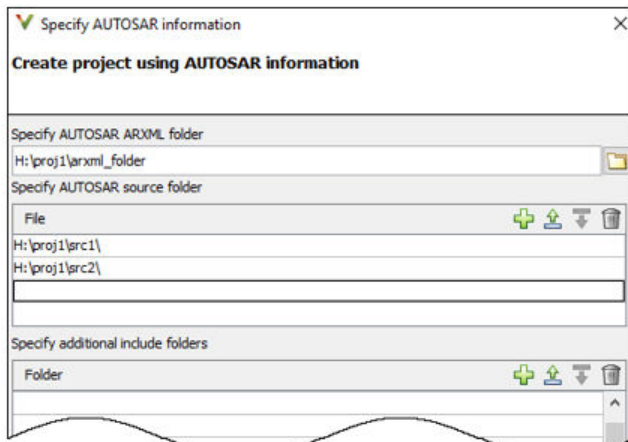
This topic shows how to run Polyspace on code implementation of AUTOSAR software components. You can run Polyspace from the user interface of the Polyspace desktop products or the command line:

- 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 <code>polyspaceroot\help\toolbox\codeprover\examples\polyspace_autosar</code> .

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 Specify the top level folder containing your ARXML files and all the folders containing source files. 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 7-5.

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 7-17.

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 the verification to be performed on a remote cluster and the results uploaded to Polyspace Metrics. Enter this advanced option:

```
-extra-project-options "-add-to-results-repository -batch -scheduler localhost"
```

Here `localhost` indicates that the same computer serves as the server and client. Replace it with the name of your server. See also [Run Bug Finder or Code Prover analysis on a remote cluster \(-batch\)](#) and [Upload results to Polyspace Metrics \(-add-to-results-repository\)](#).

- 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 8-2.

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.
- Automatic Orange Tester options: You cannot use the Automatic Orange Tester when running Polyspace on code implementation of AUTOSAR software components.

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 7-5.

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 17-78.

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 17-78.

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 7-5
- “Using Polyspace in AUTOSAR Software Development” on page 7-2
- “Run Polyspace on AUTOSAR Code with Conservative Assumptions” on page 7-21
- “Review Polyspace Results on AUTOSAR Code” on page 17-78
- “Troubleshoot Polyspace Analysis of AUTOSAR Code” on page 7-17

Troubleshoot Polyspace Analysis of AUTOSAR Code

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. If no software components are read because of errors in the ARXML, a message like this can appear.

Behaviors with Unit-Prove Environment

Summary of Actions

State after last command execution: error_fail Updating

Execution reported errors and warnings. Reported errors See detailed log messages.

Click the **See detailed log messages** link. You see the exact location of the error in the XML.

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:

- “Could Not Find Include File” on page 22-35
- “Data Type Not Recognized” on page 22-38

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 7-12
- “Conflicting Universal Unique Identifiers (UUIDs)” on page 22-37
- “Could Not Find Include File” on page 22-35
- “Data Type Not Recognized” on page 22-38

Run Polyspace on AUTOSAR Code with Conservative Assumptions

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 7-12

Run Polyspace on AUTOSAR Code Using Build Command

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 7-12

Configure Polyspace Analysis

Specify Polyspace Analysis Options

You can change the default options associated with a Polyspace analysis. For instance, you can:

- Change the set of defects that Bug Finder looks for.
See Find defects (-checkers).
- Change the default behavior of run-time checkers in Code Prover.

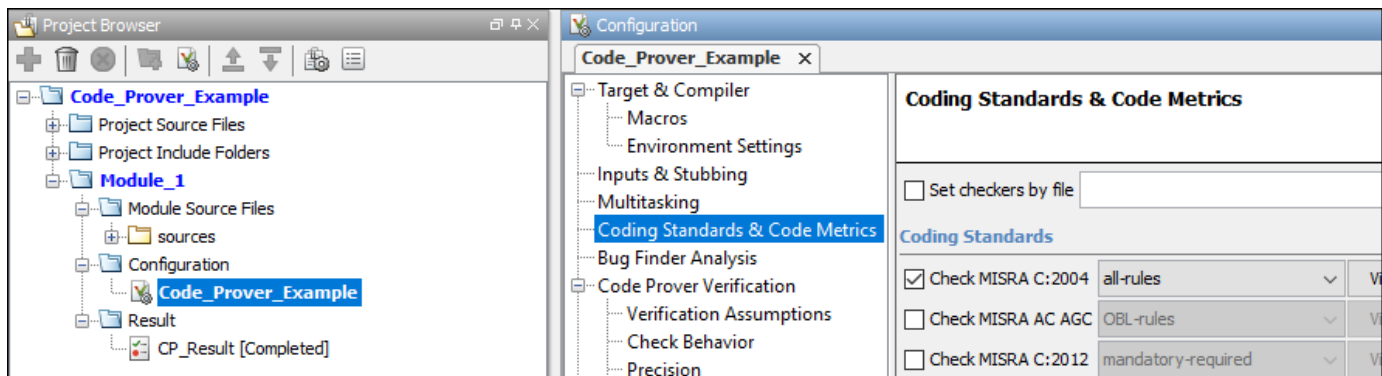
See, for instance, Overflow mode for unsigned integer (-unsigned-integer-overflows).

For the full list of analysis options, see “Analysis Options”.

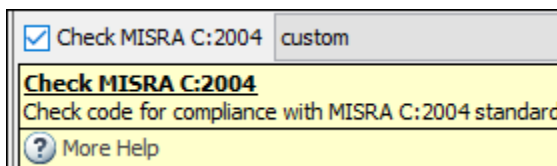
Depending on how you run Polyspace, you can configure the analysis options accordingly.

Polyspace User Interface

In the Polyspace user interface, you create a project for the analysis. The project can have one or more modules. Click the **Configuration** node in a module. On the **Configuration** pane, change options as needed.



For more information, see the tooltip on each option. Click the **More help** link for context-sensitive help on the options.



For more information, see “Run Polyspace Analysis on Desktop” on page 1-7.

Windows or Linux Scripts

Provide the options to the `polyspace-bug-finder` or `polyspace-code-prover` command. See also:

- `polyspace-bug-finder`

- `polyspace-code-prover`

For instance:

```
polyspace-code-prover -sources file_name \
    -main-generator main-generator-writes-variables all
```

You can also provide the options in a text file. See “Run Polyspace Analysis from Command Line” on page 2-2.

MATLAB Scripts

Create a `polyspace.Project` object and set the options through the `Configuration` property of the object. See also:

- `polyspace.Project`
- `polyspace.Project.Configuration` Properties

For instance:

```
proj = polyspace.Project;
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.BugFinderAnalysis.EnableCheckers = false;
```

See also “Run Polyspace Analysis by Using MATLAB Scripts” on page 3-5.

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 in Eclipse” on page 6-2.

Simulink

In your Simulink model, specify the basic options through Simulink Configuration Parameters. Select **Code > Polyspace > Options**.

From this window, you can navigate to the full set of Polyspace analysis options.

See:

- “Run Polyspace Analysis on Code Generated with Embedded Coder” on page 5-2
- “Configure Advanced Polyspace Options in Simulink” on page 5-39

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 5-56
- “Configure Advanced Polyspace Options in MATLAB Coder App” on page 5-62

Configure Target and Compiler Options

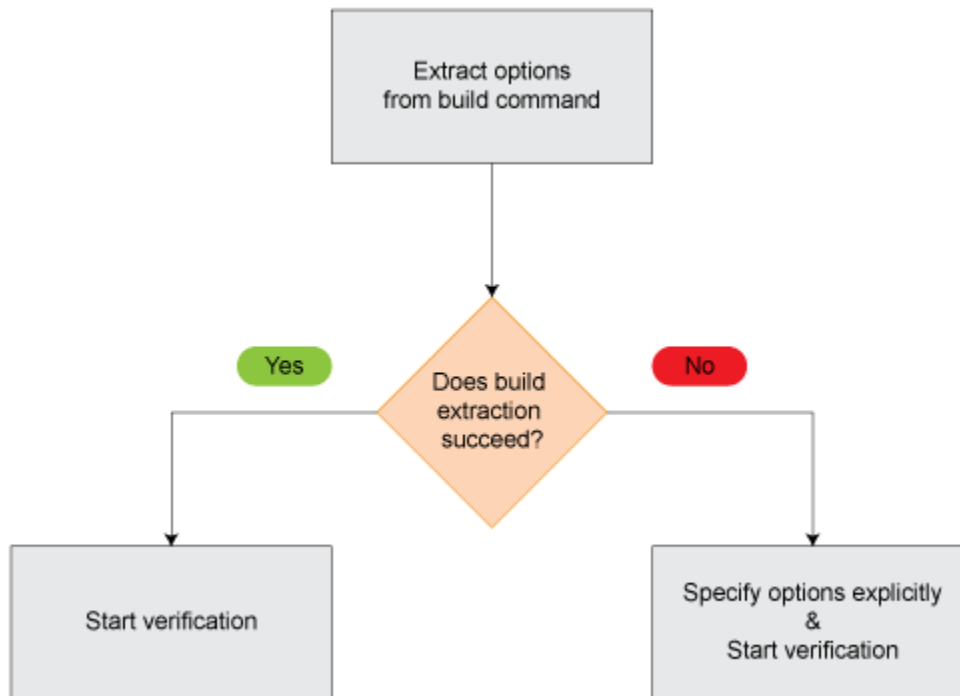
Specify Target Environment and Compiler Behavior

Before verification, specify your source code language (C or C++), target processor, and the compiler that you use for building your code. In certain cases, to emulate your compiler behavior, you might have to specify additional options.

Using your specification, the verification determines the sizes of fundamental types, considers certain macros as defined, and interprets compiler-specific extensions of the Standard. If the options do not correspond to your run-time environment, you can encounter:

- Compilation errors
- Verification results that might not apply to your target

If you use a build command such as `gmake` to build your code and the build command meets certain restrictions, you can extract the options from the build command. Otherwise, specify the options explicitly.



Extract Options from Build Command

If you use build automation scripts to build your source code, you can set up a Polyspace project from your scripts. The options associated with your compiler are specified in that project.

In the Polyspace desktop products, for information on how to trace your build command from the:

- Polyspace user interface, see "Add Source Files for Analysis in Polyspace User Interface" on page 1-2.
- DOS or UNIX command line, see `polyspace-configure`.

- MATLAB command line, see `polyspaceConfigure`.

In the Polyspace server products, for information on how to trace your build command, see “Create Polyspace Analysis Configuration from Build Command” (Polyspace Code Prover Server).

For Polyspace project creation, your build automation script (makefile) must meet certain requirements. See “Requirements for Project Creation from Build Systems” on page 9-15.

Specify Options Explicitly

If you cannot trace your build command and therefore manually create a project, you have to specify the options explicitly.

- In the user interface of the Polyspace desktop products, select a project configuration. On the **Configuration** pane, select **Target & Compiler**. Specify the options.
- At the DOS or UNIX command line, specify flags with the `polyspace-bug-finder`, `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server` command.
- At the MATLAB command line, specify arguments with the `polyspaceBugFinder`, `polyspaceCodeProver`, `polyspaceBugFinderServer` or `polyspaceCodeProverServer` function.

Specify the options in this order.

- Required options:
 - **Source code language (-lang)**: If all files have the same extension `.c` or `.cpp`, the verification uses the extension to determine the source code language. Otherwise, explicitly specify the option.
 - **Compiler (-compiler)**: Select the compiler that you use for building your source code. If you cannot find your compiler, use an option that closely matches your compiler.
 - **Target processor type (-target)**: Specify the target processor on which you intend to execute your code. For some processors, you can change the default specifications. For instance, for the processor `hc08`, you can change the size of types `double` and `long double` from 32 to 64 bits.

If you cannot find your target processor, you can create your own target and specify the sizes of fundamental types, default signedness of `char`, and endianness of the target machine. See **Generic target options**.

- Language-specific options:
 - **C standard version (-c-version)**: The default C language standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C99 standard. Specify an earlier standard such as C90 or a later standard such as C11.
 - **C++ standard version (-cpp-version)**: The default C++ language standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C++03 standard. Specify later standards such as C++11 or C++14.
- Compiler-specific options:

Whether these options are available or not depends on your specification for `Compiler (-compiler)`. For instance, if you select a `visual` compiler, the option `Pack alignment value`

(`-pack-alignment-value`) is available. Using the option, you emulate the compiler option `/Zp` that you use in Visual Studio.

For all compiler-specific options, see “Target and Compiler”.

- Advanced options:

Using these options, you can modify the verification results. For instance, if you use the option `Division round down (-div-round-down)`, the verification considers that quotients from division or modulus of negative numbers are rounded down. Use these options only if you use similar options when compiling your code.

For all advanced options, see “Target and Compiler”.

- Compiler header files:

If you specify the `diab`, `tasking` or `greenhills` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis” on page 9-14.

If you still see compilation errors after running analysis, you might have to specify other options:

- *Define macros*: Sometimes, a compilation error occurs because the analysis considers a macro as undefined. Explicitly define these macros. See `Preprocessor definitions (-D)`.
- *Specify include files*: Sometimes, a compilation error occurs because your compiler defines standard library functions differently from Polyspace and you do not provide your compiler include files. Explicitly specify the path to your compiler include files. See “Provide Standard Library Headers for Polyspace Analysis” on page 9-14.

See Also

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

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 9-5
- “Provide Standard Library Headers for Polyspace Analysis” on page 9-14

C/C++ Language Standard Used in Polyspace Analysis

The Polyspace analysis adheres to a specific language standard for code compilation. The language standard, along with your compiler specification, defines the language elements that you can use in your code. For instance, if the Polyspace analysis uses the C99 standard, C11 features such as use of the thread support library from `threads.h` causes compilation errors.

Supported Language Standards

The Polyspace analysis supports these standards:

- **C:** C90, C99, C11

The default standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C99 standard. To change the language standard, use the option `C standard version (-c-version)`.

- **C++:** C++03, C++11, C++14

The default standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C++03 standard. To change the language standard, use the option `C++ standard version (-cpp-version)`.

Default Language Standard

The default language standard depends on your specification for the option `Compiler (-compiler)`.

Compiler	C Standard	C++ Standard
generic	C99	C++03
gnu3.4, gnu4.6, gnu4.7, gnu4.8, gnu4.9	C99	C++03
gnu5.x	C11	C++03
gnu6.x	C11	C++14
gnu7.x	C11	C++14
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.
visual9.0, visual10.0, visual11.0, visual12.0	C99	C++03

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

See Also

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

More About

- “C11 Language Elements Supported in Polyspace” on page 9-7
- “C++11 Language Elements Supported in Polyspace” on page 9-8
- “C++14 Language Elements Supported in Polyspace” on page 9-11

C11 Language Elements Supported in Polyspace

This table provides a partial list of C language elements that have been introduced since C11 and the corresponding Polyspace support. If your code contains non-supported constructions, Polyspace reports a compilation error.

C11 Language Element	Supported
<code>alignas</code> and <code>alignof</code> convenience macros	Yes
<code>aligned_alloc</code> function	Yes
<code>noreturn</code> convenience macros	Yes
Generic selection	Yes
Thread support library (<code>threads.h</code>)	Yes
Atomic operations library (<code>stdatomic.h</code>)	Yes
Atomic types with <code>_Atomic</code>	Yes. If you use the Clang compiler, see limitations book for limitations on atomic data types. See “Limitations of Polyspace Verification”.
UTF-16 and UTF-32 character utilities	Yes
Bound-checking interfaces or alternative versions of standard library functions that check for buffer overflows (Annex K of C11) For instance, <code>strcpy_s</code> is an alternative to <code>strcpy</code> that checks for certain errors in the string copy.	No. Polyspace checks for certain run-time errors in use of standard library functions. The checking does not extend to these alternatives.
Anonymous structures and unions	Yes
Static assert declaration	Yes
Features related to error handling such as <code>errno_t</code> and <code>rsize_t</code> typedef-s	No. If you see compilation errors from use of these typedef-s, explicitly specify the path to your compiler headers. See “Provide Standard Library Headers for Polyspace Analysis” on page 9-14.
<code>quick_exit</code> and <code>at_quick_exit</code>	Yes. In Bug Finder, functions registered with <code>at_quick_exit</code> appear as uncalled.
<code>CMPLX</code> , <code>CMPLXF</code> and <code>CMPLXL</code> macros	Yes

See Also

C standard version (`-c-version`)

More About

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

C++11 Language Elements Supported in Polyspace

This table provides a partial list of C++ language elements that have been introduced since C++11 and the corresponding Polyspace support. If your code contains non-supported constructions, Polyspace reports a compilation error.

C++11 Std Ref	Description	Supported
C++2011-DR226	Default template arguments for function templates	Yes
C++2011-DR339	Solving the SFINAE problem for expressions	Yes
C++2011-N1610	Initialization of class objects by rvalues	Yes
C++2011-N1653	C99 preprocessor	Yes
C++2011-N1720	Static assertions	Yes
C++2011-N1737	Multi-declarator auto	Yes
C++2011-N1757	Right angle brackets	Yes
C++2011-N1791	Extended friend declarations	No
C++2011-N1811	long long	Yes
C++2011-N1984	auto-typed variables	Yes
C++2011-N1986	Delegating constructors	Yes
C++2011-N1987	Extern templates	Yes
C++2011-N1988	Extended integral types	Yes
C++2011-N2118	Rvalue references	Yes
C++2011-N2170	Universal character name literals	Yes
C++2011-N2179	Concurrency: Propagating exceptions	No
C++2011-N2235	Generalized constant expressions	Yes
C++2011-N2239	Concurrency: Sequence points	No new syntax/ keyword is introduced and therefore does not affect Polyspace support for C++11.
C++2011-N2242	Variadic templates	Yes
C++2011-N2249	New character types	Yes
C++2011-N2253	Extending sizeof	Yes
C++2011-N2258	Template aliases	Yes
C++2011-N2340	<code>__func__</code> predefined identifier	Yes
C++2011-N2341	Alignment support	Yes
C++2011-N2342	Standard Layout Types	Yes
C++2011-N2343	Declared type of an expression	Yes
C++2011-N2346	Defaulted and deleted functions	Yes
C++2011-N2347	Strongly typed enums	Yes

C++11 Std Ref	Description	Supported
C++2011-N2427	Concurrency: Atomic operations	No
C++2011-N2429	Concurrency: Memory model	No new syntax/ keyword is introduced and therefore does not affect Polyspace support for C++11.
C++2011-N2431	Null pointer constant	Yes
C++2011-N2437	Explicit conversion operators	Yes
C++2011-N2439	Rvalue references for *this	Yes
C++2011-N2440	Concurrency: Abandoning a process and at_quick_exit	Yes
C++2011-N2442	Unicode string literals	Yes
C++2011-N2442	Raw string literals	Yes
C++2011-N2535	Inline namespaces	Yes
C++2011-N2540	Inheriting constructors	Yes
C++2011-N2541	New function declarator syntax	Yes
C++2011-N2544	Unrestricted unions	Yes
C++2011-N2546	Removal of auto as a storage-class specifier	Yes
C++2011-N2547	Concurrency: Allow atomics use in signal handlers	No
C++2011-N2555	Extending variadic template template parameters	Yes
C++2011-N2657	Local and unnamed types as template arguments	Yes
C++2011-N2659	Concurrency: Thread-local storage	No
C++2011-N2660	Concurrency: Dynamic initialization and destruction with concurrency	Yes
C++2011-N2664	Concurrency: Data-dependency ordering: atomics and memory model	No
C++2011-N2672	Initializer lists	Yes
C++2011-N2748	Concurrency: Strong Compare and Exchange	No
C++2011-N2752	Concurrency: Bidirectional Fences	No
C++2011-N2756	Nonstatic data member initializers	Yes
C++2011-N2761	Generalized attributes	Yes
C++2011-N2764	Forward declarations for enums	Yes
C++2011-N2765	User-defined literals	Yes
C++2011-N2927	New wording for C++0x lambdas	Yes
C++2011-N2928	Explicit virtual overrides	Yes
C++2011-N2930	Range-based for	Yes
C++2011-N3050	Allowing move constructors to throw [noexcept]	Yes
C++2011-N3053	Defining move special member functions	Yes

C++11 Std Ref	Description	Supported
C++2011-N3276	decltype and call expressions	Yes

See Also

C++ standard version (-cpp-version)

More About

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

C++14 Language Elements Supported in Polyspace

This table provides a partial list of C++ language elements that have been introduced since C++14 and the corresponding Polyspace support. If your code contains non-supported constructions, Polyspace reports a compilation error.

C++14 Std Ref	Description	Supported
C++2014-N3323	Implicit conversion from class type in certain contexts such as <code>delete</code> or <code>switch</code> statement.	This C++14 feature allows implicit conversion from class type in certain contexts. No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3462	More SFINAE-friendly <code>std::result_of</code>	Yes
C++2014-N3472	Binary literals, for instance, <code>0b100</code> .	Yes
C++2014-N3545	<code>operator()</code> in <code>integral_constant</code> template of <code>constexpr</code> type	Yes
C++2014-N3637	Relation between <code>std::async</code> and destructor of <code>std::future</code>	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3638	Automatic deduction of return type for functions where an explicit return type is not specified	Yes. In some cases, Code Prover can show compilation errors.
C++2014-N3642	Suffixes for user-defined literals indicating time (<code>h</code> , <code>min</code> , <code>s</code> , <code>ms</code> , <code>us</code> , <code>ns</code>) and strings (<code>s</code>)	Yes
C++2014-N3648	Initialization of captured members in lambda functions	Yes. In some cases, during initialization, Code Prover can call the corresponding constructors more number of times than necessary.
C++2014-N3649	Generic (polymorphic) lambda expressions: <ul style="list-style-type: none"> Using <code>auto</code> type-specifier for parameter and return type Conversion of generic capture-less lambda expressions to pointer-to-function. 	Yes

C++14 Std Ref	Description	Supported
C++2014-N3651	Variable templates	Yes
C++2014-N3652	Declarations, conditions and loops in <code>constexpr</code> functions.	Yes
C++2014-N3653	<p>Initialization of aggregate classes with fewer initializers than members</p> <p>For instance, this initialization has fewer initializers than members. The member <code>c</code> is initialized with the value 0 and <code>d</code> is initialized with the value <code>s</code>.</p> <pre>struct S { int a; const char* b; int c; int d = b[a];}; S ss = { 1, "asdf" };</pre>	Yes
C++2014-N3654	<code>std::quoted</code>	Yes
C++2014-N3656	<code>std::make_unique</code>	Yes
C++2014-N3658	<code>std::integer_sequence</code>	Yes
C++2014-N3658	<code>std::shared_lock</code>	No. The use of <code>std::shared_lock</code> does not cause compilation errors but the construct is not semantically supported.
C++2014-N3664	Calling <code>new</code> and <code>delete</code> operators in batches.	This C++14 feature clarifies how successive calls to the <code>new</code> operator are implemented. No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3668	<code>std::exchange</code>	Partially supported.
C++2014-N3670	Using <code>std::get</code> with a data type to get one element in an <code>std::tuple</code> (provided there is only one element of the type in the tuple)	Yes
C++2014-N3671	Overloads for <code>std::equal</code> , <code>std::mismatch</code> and <code>std::is_permutation</code> function templates that accept two separate ranges	Yes
C++2014-N3733	Removal of <code>std::gets</code> from <code><cstdio></code>	Yes

C++14 Std Ref	Description	Supported
C++2014-N3776	Wording change for destructor of <code>std::future</code>	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3779	<code>std::complex</code> literals representing pure imaginary numbers with suffix <code>i</code> , <code>if</code> or <code>il</code>	Yes
C++2014-N3781	Use of single quotation mark as digit separator, for instance, <code>1'000</code> .	Yes
C++2014-N3786	Prohibiting "out of thin air" results in C++14	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3910	Synchronizing behavior of signal handlers	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3924	Discouraging use of <code>rand()</code>	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3927	Lock-free executions	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.

See Also

C++ standard version (`-cpp-version`)

More About

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

Provide Standard Library Headers for Polyspace Analysis

Before Polyspace analyzes the code for bugs and run-time errors, it compiles your code. Even if the code compiles with your compiler, you can see compilation errors with Polyspace. If the error comes from a standard library function, it usually indicates that Polyspace is not using your compiler headers. To work around the errors, provide the path to your compiler headers.

This topic shows how to locate the standard library headers from your compiler. The code examples cause a compilation error that shows the location of the headers.

- To locate the folder containing your C compiler system headers, compile this C code by using your compilation toolchain:

```
float fopen(float f);
#include <stdio.h>
```

The code does not compile because the `fopen` declaration conflicts with the declaration inside `stdio.h`. The compilation error shows the location of your compiler implementation of `stdio.h`. Your C standard library headers are all likely to be in that folder.

- To locate the folder containing your C++ compiler system headers, compile this C++ code by using your compilation toolchain:

```
namespace std {
    float cin;
}
#include <iostream>
```

The code does not compile because the `cin` declaration conflicts with the declaration inside `iostream.h`. The compilation error shows the location of your compiler implementation of `iostream.h`. Your C++ standard library headers are all likely to be in that folder.

After you locate the path to your compiler's header files, specify the path for the Polyspace analysis. For C++ code, specify the paths to both your C and C++ headers.

- In the user interface (Polyspace desktop products), add the folder to your project.
For more information, see “Add Source Files for Analysis in Polyspace User Interface” on page 1-2.
- At the command line, use the flag `-I` with the `polyspace-bug-finder`, `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server` command..

For more information, see `-I`.

See Also

More About

- “Errors from Conflicts with Polyspace Header Files” on page 22-65

Requirements for Project Creation from Build Systems

For automatic project creation from build systems, your build commands or makefiles must meet certain requirements.

Compiler Requirements

- Your compiler must be called locally.

If you use a compiler cache such as `ccache` or a distributed build system such as `distmake`, the software cannot trace your build. You must deactivate them.

- Your compiler must perform a clean build.

If your compiler performs only an incremental build, use appropriate options to build all your source files. For example, if you use `gmake`, append the `-B` or `-W makefileName` option to force a clean build. For the list of options allowed with the GNU® `make`, see `make options`.

- Your compiler configuration must be available to Polyspace. The compilers currently supported include the following:

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

If your compiler configuration is not available to Polyspace:

- Write a compiler configuration file for your compiler in a specific format. For more information, see “Compiler Not Supported for Project Creation from Build Systems” on page 22-23.
- Contact MathWorks Technical Support. For more information, see “Contact Technical Support About Issues with Running Polyspace” on page 22-18.
- If you build your code in Cygwin™, you must be using version 2.x of Cygwin for Polyspace project creation from your build system (for instance, Cygwin version 2.10).
- With the TASKING compiler, if you use an alternative `sfr` file with extension `.asfr`, Polyspace might not be able to locate your file. If you encounter an error, explicitly `#include` your `.asfr` file in the preprocessed code using the option `Include (-include)`.

Typically, you use the statement `#include __SFRFILE__(__CPU__)` along with the compiler option `--alternative-sfr-file` to specify an alternative `sfr` file. The path to the file is typically `Tasking_C166_INSTALL_DIR\include\sfr\regCPUNAME.asfr`. For instance, if your

TASKING compiler is installed in `C:\Program Files\Tasking\C166-VX_v4.0r1\` and you use the CPU-related flag `-Cxc2287m_104f` or `--cpu=xc2287m_104f`, the path is `C:\Program Files\Tasking\C166-VX_v4.0r1\include\sfr\regxc2287m.asfr`.

Build Command Requirements

- Your build command must run to completion without any user interaction.
- In Linux, only UNIX shell (sh) commands must be used. If your build uses advanced commands such as commands supported only by bash, tcsh or zsh, Polyspace cannot trace your build.

In Windows, only DOS commands must be used. If your build uses advanced commands such as commands supported only by PowerShell or Cygwin, Polyspace cannot trace your build. To see if Polyspace supports your build command, run the command from `cmd.exe` in Windows. For more information, see “Check if Polyspace Supports Build Scripts” on page 22-30.

- If you use statically linked libraries, Polyspace cannot trace your build. In Linux, you can install the full Linux Standard Base (LSB) package to allow dynamic linking. For example, on Debian® systems, install LSB with the command `apt-get install lsb`.
- Your build command must not use aliases.

The `alias` command is used in Linux to create an alternate name for commands. If your build command uses those alternate names, Polyspace cannot recognize them.

- Your build process must not use the `LD_PRELOAD` mechanism.
- Your build command must be executable completely on the current machine and must not require privileges of another user.

If your build uses `sudo` to change user privileges or `ssh` to remotely log in to another machine, Polyspace cannot trace your build.

- If your build command uses redirection with the `>` or `|` character, the redirection occurs after Polyspace traces the command. Therefore, Polyspace does not handle the redirection.

For example, if your command occurs as

```
command1 | command2
```

And you enter

```
polyspace-configure command1 | command2
```

When tracing the build, Polyspace traces the first command only.

- If the System Integrity Protection (SIP) feature is active on the operating system macOS El Capitan (10.11) or a later macOS version, Polyspace cannot trace your build command. Before tracing your build command, disable the SIP feature. You can 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.

Note Your environment variables are preserved when Polyspace traces your build command.

See Also

polyspace-configure

Related Examples

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

Emulate Microchip MPLAB XC16 and XC32 Compilers

If you build your source code using Microchip MPLAB XC16 or XC32 compilers, you can set up your Polyspace analysis so that your code will compile with Polyspace. Enter these options at the command line or specify them in the **Configuration** pane of the Polyspace desktop user interface.

Compiler	Target Processor Families	Options
MPLAB XC16	PIC24 dsPIC	-compiler gnu4.6 -to compile -D __XC__ -D __XC16__ -target=mcpu -wchar-t-type-is unsigned-int -align 16 -long-long-is-64bits
MPLAB XC32	PIC32	-compiler gnu4.8 -custom-target true,8,2,4,-1,4,8,4, 4,8,4,8,1,big,unsigned_long,long,i -D __PIC32M -D __PIC32MX -D __PIC32MX__ -D __XC32 -D __XC32__ -D __XC__ -D __XC__ -D __mips=32 -D __mips__ -D __mips

The set of macros specified with the option Preprocessor definitions (-D) is a minimal set. Specify additional macros as needed to ensure your code compiles with Polyspace.

See Also

-custom-target | Generic target options

More About

- “Specify Polyspace Analysis Options” on page 8-2
- “Specify Target Environment and Compiler Behavior” on page 9-2

Supported Keil or IAR Language Extensions

Polyspace analysis can interpret a subset of common C/C++ language constructs and extended keywords by default. For compiler-specific keywords, you must specify your choice of compiler. If you specify `keil` or `iar` for `Compiler` (`-compiler`), the Polyspace verification allows language extensions specific to the Keil or IAR compilers.

Special Function Register Data Type

Embedded control applications frequently read and write port data, set timer registers, and read input captures. To deal with these requirements without using assembly language, some microprocessor compilers define special data types such as `sfr` and `sbit`. Typical declarations are:

```
sfr A0 = 0x80;
sfr A1 = 0x81;
sfr ADCUP = 0xDE;
sbit EI = 0x80;
```

The declarations reside in header files such as `regxx.h` for the basic 80Cxxx micro processor. The declarations customize the compiler to the target processor.

You access a register or a port by using the `sfr` and `sbit` data as follows. However, these data types are not part of the C99 Standard.

```
int status,P0;

void main (void) {
    ADCUP = 0x08; /* Write data to register */
    A1 = 0xFF; /* Write data to Port */
    status = P0; /* Read data from Port */
    EI = 1; /* Set a bit (enable all interrupts) */
}
```

To analyze this type of code, use these options:

- `Compiler` (`-compiler`): Specify `keil` or `iar`.
- `Sfr type support` (`-sfr-types`): Specify the data type and size in bits.

For example, depending on how you define the `sbit` data type, you use these options:

- `sbit ADST = ADCUP^7;`
Use options: `-compiler keil -sfr-type sfr=8`
- `sbit ADST = ADCUP.7;`
Use options: `-compiler iar -sfr-type sfr=8`

The analysis then supports the Keil or IAR language extensions even if some structures, keywords, and syntax are not part of the C99 standard.

Keywords Removed During Preprocessing

Once you specify the Keil or IAR compiler, the analysis recognizes compiler-specific keywords in your code. If a keyword is not relevant for the analysis, it is removed from the source code during preprocessing.

If you disable the keyword and use it as an identifier instead, you can encounter a compilation error when you compile your code with Polyspace. See “Errors Related to Keil or IAR Compiler” on page 22-48.

These keywords are removed during preprocessing:

- Keil: `bdata`, `far`, `idata`, `huge`, `sdata`
- IAR: `saddr`, `reentrant`, `reentrant_idata`, `non_banked`, `plm`, `bdata`, `idata`, `pdata`, `code`, `xdata`, `xhuge`, `interrupt`, `__interrupt`, `__intrinsic`

The `data` keyword is not removed.

Remove or Replace Keywords Before Compilation

The Polyspace compiler strictly follows the ANSI® C99 Standard (ISO/IEC 9899:1999). If your compiler allows deviation from the Standard, the Polyspace compilation using default options cannot emulate your compiler. For instance, your compiler can allow certain non-ANSI keyword, which Polyspace does not recognize by default.

To emulate your compiler closely, you specify the Target & Compiler options. If you still get compilation errors from unrecognized keywords, you can remove or replace them only for the purposes of verification. The option `Preprocessor definitions (-D)` allows you to make simple substitutions. For complex substitutions, for instance to remove a group of space-separated keywords such as a function attribute, use the option `Command/script` to apply to preprocessed files (`-post-preprocessing-command`).

Remove Unrecognized Keywords

You can remove unsupported keywords from your code for the purposes of analysis. For instance, follow these steps to remove the `far` and `0x` keyword from your code (`0x` precedes an absolute address).

- 1 Save the following template as `C:\Polyspace\myTpl.pl`.

Content of myTpl.pl

```
#!/usr/bin/perl

#####
# Post Processing template script
#
#####
# Usage from GUI:
#
# 1) Linux: /usr/bin/perl PostProcessingTemplate.pl
# 2) Windows: polyspaceroot\sys\perl\win32\bin\perl.exe <pathtoscript>\
# PostProcessingTemplate.pl
#
#####

$version = 0.1;

$INFILE = STDIN;
$OUTFILE = STDOUT;

while (<$INFILE>)
{
    # Remove far keyword
    s/far//;

    # Remove "@ 0xFE1" address constructs
    s/\s0x[A-F0-9]*//g;

    # Remove "@0xFE1" address constructs
    s/\s@[A-F0-9]*//g;

    # Remove "@ ((unsigned)&LATD*8)+2" type constructs
    s/\s\(\(unsigned\)&[A-Z0-9]+\*8\)\+\d//g;


    # DON'T DELETE LINE BELOW: Print the current processed line
    print $OUTFILE $_;
}
```

For reference, see a summary of Perl regular expressions.

Perl Regular Expressions

```
#####
# Metacharacter What it matches
```

```
#####
# Single Characters
# . Any character except newline
# [a-z0-9] Any single character in the set
# [^a-z0-9] Any character not in set
# \d A digit same as
# \D A non digit same as [^0-9]
# \w An Alphanumeric (word) character
# \W Non Alphanumeric (non-word) character
#
# Whitespace Characters
# \s Whitespace character
# \S Non-whitespace character
# \n newline
# \r return
# \t tab
# \f formfeed
# \b backspace
#
# Anchored Characters
# \B word boundary when no inside []
# \B non-word boundary
# ^ Matches to beginning of line
# $ Matches to end of line
#
# Repeated Characters
# x? 0 or 1 occurrence of x
# x* 0 or more x's
# x+ 1 or more x's
# x{m,n} Matches at least m x's and no more than n x's
# abc All of abc respectively
# to|be|great One of "to", "be" or "great"
#
# Remembered Characters
# (string) Used for back referencing see below
# \1 or $1 First set of parentheses
# \2 or $2 First second of parentheses
# \3 or $3 First third of parentheses
#####
# Back referencing
#
# e.g. swap first two words around on a line
# red cat -> cat red
# s/(\w+) (\w+)/$2 $1/;
#
#####
```

- 2 On the **Configuration** pane, select **Environment Settings**.
- 3 To the right of **Command/script to apply to preprocessed files**, click .
- 4 Use the Open File dialog box to navigate to C:\Polyspace.
- 5 In the **File name** field, enter myTpl.pl.
- 6 Click **Open**. You see C:\Polyspace\myTpl.pl in the **Command/script to apply to preprocessed files** field.

Remove Unrecognized Function Attributes

You can remove unsupported function attributes from your code for the purposes of analysis.

If you run verification on this code specifying a generic compiler, you can see compilation errors from the `noreturn` attribute. The code compiles using a GNU compiler.

```
void fatal () __attribute__ ((noreturn));

void fatal (/* ... */)
{
    /* ... */ /* Print error message. */ /* ... */
    exit (1);
}
```

If the software does not recognize an attribute and the attribute does not affect the code analysis, you can remove it from your code for the purposes of verification. For instance, you can use this Perl script to remove the `noreturn` attribute.

```
while ($line = <STDIN>)
{
    # __attribute__ ((noreturn))

    # Remove far keyword
    $line =~ s/ __attribute__ \ \(\(noreturn\)\)//g;

    # Print the current processed line to STDOUT
    print $line;
}
```

Specify the script using the option `Command/script` to apply to preprocessed files (`-post-preprocessing-command`).

See Also

Polyspace Analysis Options

`Command/script` to apply to preprocessed files (`-post-preprocessing-command`) |
Preprocessor definitions (`-D`)

Related Examples

- “Troubleshoot Compilation Errors”

Gather Compilation Options Efficiently

Polyspace verification can sometimes stop in the compilation or linking phase due to the following reasons:

- The Polyspace compiler strictly follows a C or C++ Standard (depending on your choice of compiler). See “C/C++ Language Standard Used in Polyspace Analysis” on page 9-5. If your compiler allows deviation from the Standard, the Polyspace compilation using default options cannot emulate your compiler.
- Your compiler declares standard library functions with argument or return types different from the standard types. Unless you also provide the function definition, for efficient verification, Polyspace uses its own definitions of standard library functions, which have the usual prototype. The mismatch in types causes a linking error.

You can easily work around the compilation and standard library function errors. To work around the errors, you typically specify certain analysis options. In some cases, you might have to add a few lines to your code. For instance:

- To emulate your compiler behavior more closely, you specify the Target & Compiler options. If you still face compilation errors, you might have to remove or replace certain unrecognized keywords using the option `Preprocessor definitions (-D)`. However, the option allows only simple substitution of a string with another string. For more complex replacements, you might have to add `#define` statements to your code.
- To avoid errors from stubbing standard library functions, you might have to `#define` certain Polyspace-specific macros so that Polyspace does not use its own definition of standard library functions.

Instead of adding these modifications to your original code, create a single `polyspace.h` file that contains all modifications. Use the option `Include (-include)` to force inclusion of the `polyspace.h` file in all source files under verification.

Benefits of this approach include:

- The error detection is much faster since it will be detected during compilation rather than in the link or subsequent phases.
- There will be no need to modify original source files.
- The file is automatically included as the very first file in the original `.c` files.
- The file is reusable for other projects developed under the same environment.

Example 9.1. Example

This is an example of a file that can be used with the option `Include (-include)`.

```
// The file may include (say) a standard include file implicitly
// included by the cross compiler

#include <stdlib.h>
#include "another_file.h"

// Workarounds for compilation errors
#define far
#define at(x)
```

```
// Workarounds for errors due to redefining standard library functions

#define POLYSPACE_NO_STANDARD_STUBS // use this flag to prevent the
    //automatic stubbing of std functions
#define __polyspace_no_sscanf
#define __polyspace_no_fgetc
void sscanf(int, char, char, char, char, char);
void fgetc(void);
```

See Also

More About

- “Troubleshoot Compilation Errors”

Configure Inputs and Stubbing Options

Specify External Constraints

This example shows how to specify constraints (also known as data range specifications or DRS) on variables in your code. Polyspace uses the code that you provide to make assumptions about items such as variable ranges and allowed buffer size for pointers. Sometimes the assumptions are broader than what you expect because:

- You have not provided the complete code. For example, you did not provide some of the function definitions.
- Some of the information about variables is available only at run time. For example, some variables in your code obtain values from the user at run time.

Because of these broad assumptions:

- Code Prover can consider more execution paths than those paths that occur at run time. If an operation fails along one of the execution paths, Polyspace places an orange check on the operation. If that execution path comes from an assumption that is too broad, the orange check might indicate a false positive.
- Bug Finder can sometimes produce false positives.

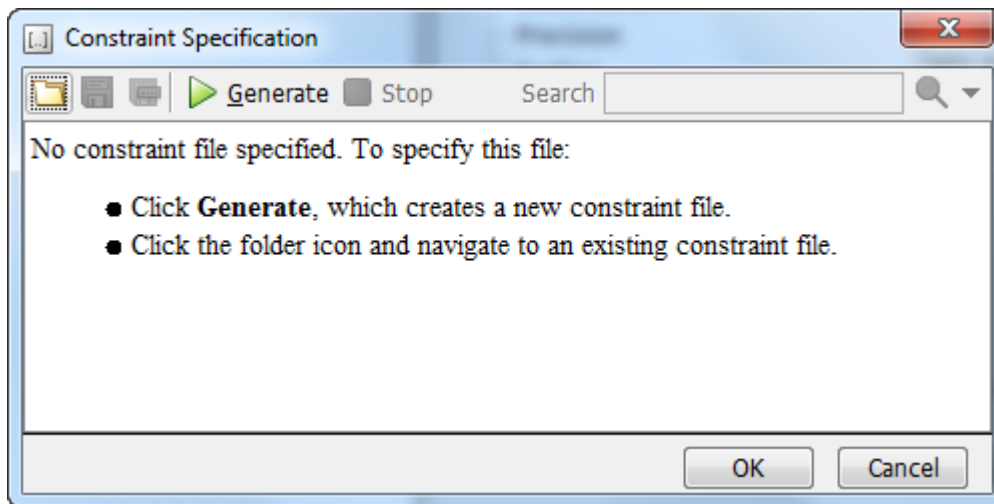
To reduce the number of such false positives, you can specify additional constraints on global variables, function inputs, and return values and modifiable arguments of stubbed functions. You save the constraints as an XML file to use them for subsequent analyses. If your source code changes, you can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

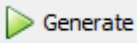
Note In Bug Finder, you can only constrain global variables. You cannot constrain function inputs or return values of stubbed functions.

Create Constraint Template

User Interface (Desktop Products Only)

- 1 Open the project configuration. On the **Configuration** pane, select **Inputs & Stubbing**.
- 2 To the right of **Constraint setup**, click the **Edit** button to open the **Constraint Specification** window.



- 3 In the Constraint Specification dialog box, create a blank constraint template. The template contains a list of all variables on which you can provide constraints. To create a new template, click . The software compiles your project and creates a template. The new template is stored in a file `Module_number_Project_name_drs_template.xml` in your project folder.
- 4 Specify your constraints and save the template as an XML file. For more information, see “External Constraints for Polyspace Analysis” on page 10-7.
- 5 Click **OK**.

You see the full path to the template XML file in the **Constraint setup** field. If you run an analysis, Polyspace uses this template for extracting variable constraints.

Command Line

Use the option `Constraint setup (-data-range-specifications)` to specify the constraints XML file.

To specify constraints in the XML file:

- 1 First, create a blank XML template. The template lists all global variables, function inputs and modifiable arguments and return values of stubbed functions without specifying any constraints on them.

To create a blank template, run an analysis only up to the compilation phase. In Bug Finder, disable checking of defects. Use the option `Find defects (-checkers)`. In Code Prover, check for source compliance only. Use the argument `compile` for the option `Verification level (-to)`. After the analysis, a blank template XML `drs-template.xml` is created in the results folder.

For C++ projects, to create a blank constraints template, you have to use the argument `cpp-normalize` for the option `Verification level (-to)`.

- 2 Edit the XML file to specify your constraints.


For examples, see:

- “Constrain Global Variable Range” on page 10-13
- “Constrain Function Inputs” on page 10-16

Create Constraint Template from Code Prover Analysis Results

You can constrain variable ranges based on their expected range in real-world applications. For instance, if a variable represents vehicle speed, you can set a maximum possible value. You can also constrain variable ranges only if they cause too many orange checks from overapproximation.

A Code Prover analysis shows all global variables, function inputs and stubbed functions that lead to orange checks from possible overapproximation. You can constrain only these variables for a more precise analysis.

- 1 Open Code Prover results in the Polyspace user interface or Polyspace Access web interface.
- 2 Open the **Orange Sources** pane. Do one of the following:
 - Select an orange check. If the software can trace an orange check to a root cause, a  icon appears on the **Result Details** pane. Click this icon to open the **Orange Sources** pane.
 - In the Polyspace user interface, select **Window > Show/Hide View > Orange Sources**. In the Polyspace Access web interface, select **Layout > Show/Hide View > Orange Sources**.

You see the full list of variables (function inputs or return values of stubbed functions) that can cause orange checks. Constrain the ranges of these variables.

In the details for individual orange checks, you often see a message similar to this:

```
If appropriate, applying DRS to stubbed function random_float in example.c  
line 44 may remove this orange.
```

The message is an indication that the stubbed function is a possible source of the orange check. You can apply external constraints on the function to enforce more precise assumptions and possibly remove the orange check (in case it came from the broader assumptions).


Update Existing Template

With new code submissions, you might have to specify additional constraints. You can update an existing template to add global variables, function inputs and stubbed functions that come from the new code submissions.

Additionally, if you remove some variables or functions from your code, constraints on them are not applicable any more. Instead of regenerating a constraint template and respecifying the constraints, you can update an existing template and remove the variables that are not present in your code.

User Interface (Desktop Products Only)

- 1 On the **Configuration** pane, select **Inputs & Stubbing**.

- 2 Open the existing template in one of the following ways:
 - In the **Constraint setup** field, enter the path to the template XML file. Click **Edit**.
 - Click **Edit**. In the Constraint Specification dialog box, click the  icon to navigate to your template file.
- 3 Click **Update**.
 - a Variables that are no longer present in your source code appear under the **Non Applicable** node. To remove an entry under the **Non Applicable** node or the node itself, right-click and select **Remove This Node**.
 - b Specify your new constraints for any of the other variables.

Command Line

In a continuous integration workflow, you can use the constraints XML file from the previous run. If new code submissions require additional constraints:

- 1 Specify constraints on variables from new code submissions in a constraints XML file. See Create Constraint Template: Command Line on page 10-3.
- 2 Merge the constraints XML file with the new constraints and the constraints XML file from the previous run.

Specify Constraints in Code

Specifying constraints outside your code allows for more precise analysis. However, you must use the code within the specified constraints because the constraints are *outside* your code. Otherwise, the results might not apply. For example, if you use function inputs outside your specified range, a run-time error can occur on an operation even though checks on the operation are green.

To specify constraints *inside* your code, you can use:

- Appropriate error handling tests in your code.

Polyspace checks to determine if the errors can actually occur. If they do not occur, the test blocks appear as **Unreachable code**.
- The `assert` macro. For example, to constrain a variable `var` in the range `[0,10]`, you can use `assert(var >= 0 && var <=10);`.

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

See Also

Constraint setup (-data-range-specifications)

Related Examples

- “External Constraints for Polyspace Analysis” on page 10-7
- “Constrain Global Variable Range” on page 10-13
- “Constrain Function Inputs” on page 10-16
- “XML File Format for Constraints” on page 10-19

External Constraints for Polyspace Analysis

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

- Global Variables.
- User-defined Functions.

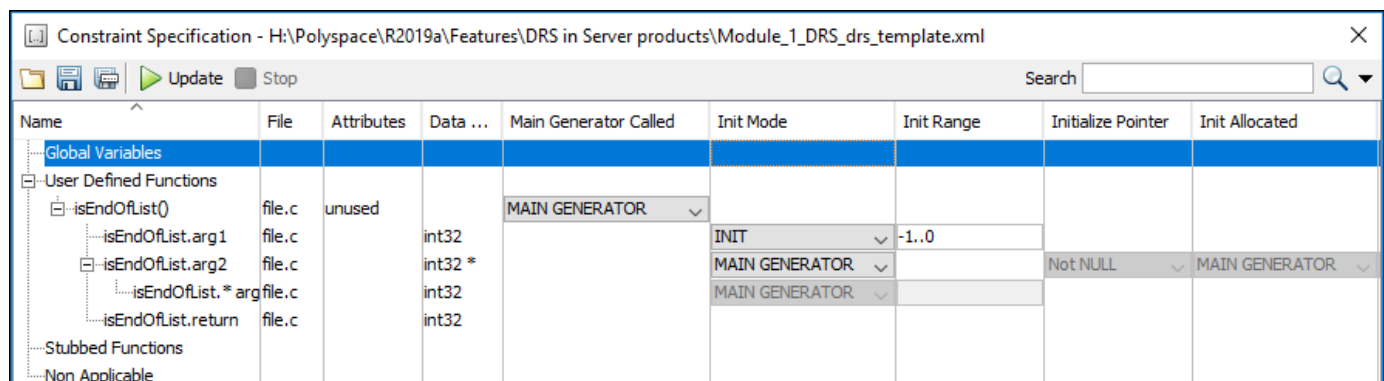
Constraints on user-defined functions do not apply to a Bug Finder analysis.

- Stubbed Functions.

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

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

In the user interface of the Polyspace desktop products, you can specify the constraints through a **Constraint Specification** window. The constraints are saved in an XML file that you can reuse for other projects.



The screenshot shows the 'Constraint Specification' window for the file 'H:\Polyspace\R2019a\Features\DRS in Server products\Module_1_DRS_drs_template.xml'. The window contains a table with the following columns: Name, File, Attributes, Data ..., Main Generator Called, Init Mode, Init Range, Initialize Pointer, and Init Allocated. The table is organized into sections: Global Variables, User Defined Functions, Stubbed Functions, and Non Applicable. Under 'User Defined Functions', there is a tree view for 'isEndOfList()' with sub-entries 'isEndOfList.arg1', 'isEndOfList.arg2', 'isEndOfList.* arg', and 'isEndOfList.return'. The table rows show constraints for these functions, with 'MAIN GENERATOR' selected in the 'Main Generator Called' column and 'INIT' selected in the 'Init Mode' column for 'isEndOfList.arg1'. The 'Init Range' for 'isEndOfList.arg1' is '-1..0'. The 'Initialize Pointer' for 'isEndOfList.* arg' is 'Not NULL' and the 'Init Allocated' is 'MAIN GENERATOR'.

Name	File	Attributes	Data ...	Main Generator Called	Init Mode	Init Range	Initialize Pointer	Init Allocated
Global Variables								
User Defined Functions								
isEndOfList()	file.c	unused		MAIN GENERATOR				
isEndOfList.arg1	file.c		int32		INIT	-1..0		
isEndOfList.arg2	file.c		int32 *		MAIN GENERATOR		Not NULL	MAIN GENERATOR
isEndOfList.* arg	file.c		int32		MAIN GENERATOR			
isEndOfList.return	file.c		int32					
Stubbed Functions								
Non Applicable								

This table explains the various columns in the **Constraint Specification** window. If you directly edit the constraint XML file to specify a constraint (for instance, in the Polyspace Server products), this table also shows the correspondence between columns in the user interface and entries in the XML file. The XML entry highlighted in bold appears in the corresponding column of the **Constraint Specification** window.

Column	Settings
Name	<p>Displays the list of variables and functions in your Project for which you can specify data ranges.</p> <p>This Column displays three expandable menu items:</p> <ul style="list-style-type: none"> • Globals - Displays global variables in the project. • User defined functions - Displays user-defined functions in the project. Expand a function name to see its inputs. • Stubbed functions - Displays a list of stub functions in the project. Expand a function name to see the inputs and return values. <p>XML File Entry:</p> <pre><function name = "funcName" ...> <scalar name = "arg1" ...> <pointer name = "arg2" ...></pre>
File	<p>Displays the name of the source file containing the variable or function.</p> <p>XML File Entry:</p> <pre><file name = "C:\Project1\Sources\file.c" ...></pre>
Attributes	<p>Displays information about the variable or function.</p> <p>For example, static variables display <code>static</code>. Uncalled functions display <code>unused</code>.</p> <p>XML File Entry:</p> <pre><function name="funcName" attributes="unused" ...></pre>
Data Type	<p>Displays the variable type.</p> <p>XML File Entry:</p> <pre><scalar name="arg1" complete_type="int32" ...></pre>
Main Generator Called	<p>Applicable only for user-defined functions.</p> <p>Specifies whether the main generator calls the function:</p> <ul style="list-style-type: none"> • MAIN GENERATOR - Main generator may call this function, depending on the value of the <code>-functions-called-in-loop</code> (C) or <code>-main-generator-calls</code> (C++) parameter. • NO - Main generator will not call this function. • YES - Main generator will call this function. <p>XML File Entry:</p> <pre><function name="funcName" main_generator_called="MAIN_GENERATOR" ...></pre>

Column	Settings
Init Mode	<p>Specifies how the software assigns a range to the variable:</p> <ul style="list-style-type: none"> • MAIN GENERATOR - Variable range is assigned depending on the settings of the main generator options <code>-main-generator-writes-variables</code> and <code>-no-def-init-glob</code>. • IGNORE - Variable is not assigned to any range, even if a range is specified. • INIT - Variable is assigned to the specified range only at initialization, and keeps the range until first write. • PERMANENT - Variable is permanently assigned to the specified range. If the variable is assigned outside this range during the program, no warning is provided. Use the <code>globalassert</code> mode if you need a warning. <p>User-defined functions support only INIT mode.</p> <p>Stub functions support only PERMANENT mode.</p> <p>For C verifications, global pointers support MAIN GENERATOR, IGNORE, or INIT mode.</p> <ul style="list-style-type: none"> • MAIN GENERATOR - Pointer follows the options of the main generator. • IGNORE - Pointer is not initialized • INIT - Specify if the pointer is NULL, and how the pointed object is allocated (Initialize Pointer and Init Allocated options). <p>XML File Entry:</p> <pre><scalar name="arg1" init_mode="INIT" ...></pre>
Init Range	<p>Specifies the minimum and maximum values for the variable.</p> <p>You can use the keywords <code>min</code> and <code>max</code> to denote the minimum and maximum values of the variable type. For example, for the type <code>long</code>, <code>min</code> and <code>max</code> correspond to -2^{31} and $2^{31}-1$ respectively.</p> <p>You can also use hexadecimal values. For example: <code>0x12..0x100</code></p> <p>For <code>enum</code> variables, you cannot specify ranges directly using the enumerator constants. Instead use the values represented by the constants.</p> <p>For <code>enum</code> variables, you can also use the keywords <code>enum_min</code> and <code>enum_max</code> to denote the minimum and maximum values that the variable can take. For example, for an <code>enum</code> variable of the type defined below, <code>enum_min</code> is 0 and <code>enum_max</code> is 5:</p> <pre>enum week{ sunday, monday=0, tuesday, wednesday, thursday, friday, saturday};</pre> <p>XML File Entry:</p> <pre><scalar name="arg1" init_range="-1..0" ...></pre>

Column	Settings
Initialize Pointer	<p>Applicable only to pointers. Enabled only when you specify Init Mode:INIT.</p> <p>Specifies whether the pointer should be NULL:</p> <ul style="list-style-type: none"> • May-be NULL - The pointer could potentially be a NULL pointer (or not). • Not Null - The pointer is never initialized as a null pointer. • Null - The pointer is initialized as NULL. <hr/> <p>Note Not applicable for C++ projects. See “Constraint Specification Limitations” on page 10-11.</p> <hr/> <p>XML File Entry:</p> <pre><pointer name="arg1" initialize_pointer="Not NULL" ...></pre>
Init Allocated	<p>Applicable only to pointers. Enabled only when you specify Init Mode:INIT.</p> <p>Specifies how the pointed object is allocated:</p> <ul style="list-style-type: none"> • MAIN GENERATOR - The pointed object is allocated by the main generator. • None - Pointed object is not written. • SINGLE - Write the pointed object or the first element of an array. (This setting is useful for stubbed function parameters.) • MULTI - All objects (or array elements) are initialized. <hr/> <p>Note Not applicable for C++ projects. See “Constraint Specification Limitations” on page 10-11.</p> <hr/> <p>XML File Entry:</p> <pre><pointer name="arg1" init_pointed="MAIN_GENERATOR" ...></pre>

Column	Settings
# Allocated Objects	<p>Applicable only to pointers.</p> <p>Specifies how many objects are pointed to by the pointer (the pointed object is considered as an array).</p> <p>The Init Allocated parameter specifies how many allocated objects are actually initialized. For instance, consider this code:</p> <pre>void func(int *ptr) { assert(ptr[0]==1); assert(ptr[1]==1); }</pre> <p>If you specify these constraints:</p> <ul style="list-style-type: none"> ptr has Init Allocated set to MULTI and # Allocated Objects set to 2, *ptr has Init Range set to 1..1, <p>both assertions are green. However, if you specify these constraints:</p> <ul style="list-style-type: none"> ptr has Init Allocated set to SINGLE *ptr has Init Range set to 1..1, <p>the second assertion is orange. Only the first object that ptr points to initialized to 1. Objects beyond the first can be potentially full range.</p> <hr/> <p>Note Not applicable for C++ projects. See “Constraint Specification Limitations” on page 10-11.</p> <hr/> <p>XML File Entry:</p> <pre><pointer name="arg1" number_allocated="10" ...></pre>
Global Assert	<p>Specifies whether to perform an assert check on the variable at global initialization, and after each assignment.</p> <hr/> <p>XML File Entry:</p> <pre><scalar name="glob" global_assert="YES" ...></pre>
Global Assert Range	<p>Specifies the minimum and maximum values for the range you want to check.</p> <hr/> <p>XML File Entry:</p> <pre><scalar name="glob" assert_range="0..200" ...></pre>
Comment	<p>Remarks that you enter, for example, justification for your DRS values.</p> <hr/> <p>XML File Entry:</p> <pre><scalar name="glob" comment="Speed Range" ...></pre>

Constraint Specification Limitations

You cannot specify these constraints:

- *C++ Pointers cannot be constrained:*

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

Because of polymorphism, a C++ pointer can point to objects of multiple classes in a class hierarchy and can require invoking different constructors. The pre-analysis for constraint specification cannot determine which object type to constrain or which constructor to call.

- *Constraints cannot be relations:*

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

- *Multiple ranges not possible:*

You cannot specify multiple ranges for a constraint. For instance, you cannot specify that a function argument has either the value -1 or a value in the range [1,100]. Instead, specify the range [-1,100] or perform two separate analyses, once with the value -1 and once with the range [1,100].

See Also

More About


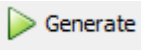
- “Specify External Constraints” on page 10-2

Constrain Global Variable Range

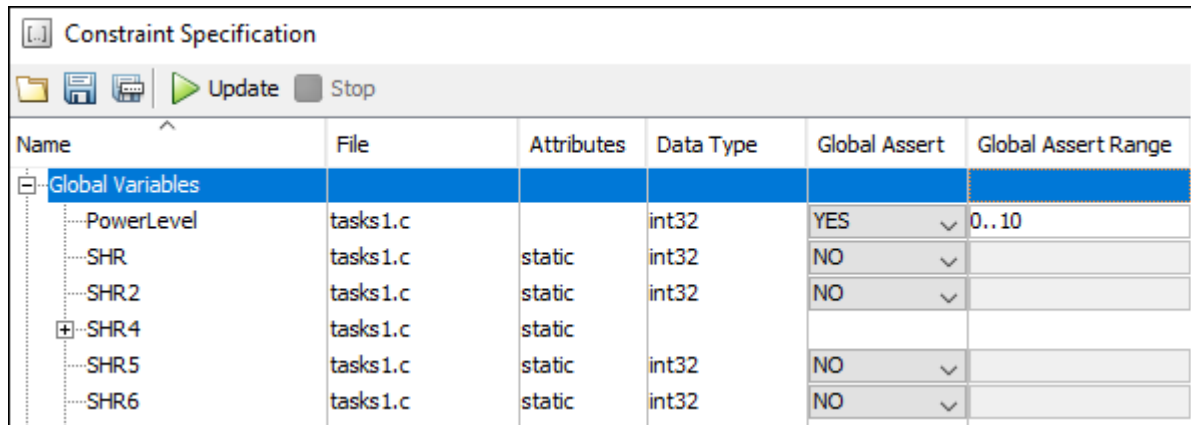
You can impose constraints (also known as data range specifications or DRS) on the range of a global variable and check with Code Prover whether write operations on the variable violate the constraint. For the general workflow, see “Specify External Constraints” on page 10-2.

User Interface (Desktop Products Only)


To constrain a global variable range and also check for violation of the constraint:

- 1 In your project configuration, select **Inputs & Stubbing**. Click the  button next to the **Constraint setup** field.
- 2 In the Constraint Specification window, click .

Under the **Global Variables** node, you see a list of global variables.



Name	File	Attributes	Data Type	Global Assert	Global Assert Range
Global Variables					
PowerLevel	tasks1.c		int32	YES	0..10
SHR	tasks1.c	static	int32	NO	
SHR2	tasks1.c	static	int32	NO	
SHR4	tasks1.c	static			
SHR5	tasks1.c	static	int32	NO	
SHR6	tasks1.c	static	int32	NO	

- 3 For the global variable that you want to constrain:
 - From the drop-down list in the **Global Assert** column, select YES.
 - In the **Global Assert Range** column, enter the range in the format *min*..*max*. *min* is the minimum value and *max* the maximum value for the global variable.
 - 4 To save your specifications, click the  button.
- In **Save a Constraint File** window, save your entries as an xml file.
- 5 Run a verification and open the results.

For every write operation on the global variable, you see a green, orange, or red **Correctness condition** check. If the check is:

- Green, the variable is within the range that you specified.
- Orange, the variable can be outside the range that you specified.
- Red, the variable is outside the range that you specified.

When two or more tasks write to the same global variable, the **Correctness condition** check can appear orange on all write operations to the variable even when only one write operation takes the variable outside the **Global Assert** range.

Command Line

Use the option `Constraint setup (-data-range-specifications)` with an XML file specifying your constraint.

For instance, for an analysis with Polyspace Code Prover Server, specify the option as follows:

```
polyspace-code-prover-server -sources filename
                             -data-range-specifications "C:\Polyspace\drs_project1.xml"
```

Create a blank constraint XML template as described in “Specify External Constraints” on page 10-2. In the XML file, locate and constrain the global variables. XML tags for global variables appear directly within the `file` tag without an enclosing function tag. For instance, in this constraint XML, `PowerLevel` and `SHR` are global variables:

```
<file name="\\\\home\\Polyspace_Workspace\\Examples\\Code_Prover_Example
          \\sources\\tasks1.c">
  <scalar name="PowerLevel" line="26" .. global_assert="YES" assert_range="0..10"/>
  <scalar name="SHR" line="30" ... global_assert="NO" assert_range="" />
  <function name="Tserver" line="73" .../>
  <function name="initregulate" line="47" .../>
  <function name="orderregulate" line="35" ...>
    <scalar name="return" ... global_assert="unsupported" assert_range="unsupported" />
  </function>
  <function name="procl" line="101" .../>
</file>
```

To specify a constraint on a global variable and check during a Code Prover analysis if the constraint is violated:

- 1 Set the `global_assert` attribute of the variable's `scalar` tag to `YES`.
- 2 Set the `assert_range` attribute to a range in the form `min..max`, for instance, `0..10`.

In the preceding example, the variable `PowerLevel` is constrained this way.

See Also

Polyspace Analysis Options

`Constraint setup (-data-range-specifications)`

Polyspace Results

Correctness condition

More About

- “Specify External Constraints” on page 10-2
- “External Constraints for Polyspace Analysis” on page 10-7
- “Constrain Function Inputs” on page 10-16

Constrain Function Inputs


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

For instance, for a function defined as follows, you can specify that the argument `val` has values in the range `[1..10]`. You can also specify that the argument `ptr` points to a 3-element array where each element is initialized:

```
int func(int val, int* ptr) {
    .
    .
}
```

User Interface (Desktop Products Only)

To specify constraints on function inputs:

- 1 In your project configuration, select **Inputs & Stubbing**. Click the  button for **Constraint setup**.

- 2 In the Constraint Specification window, click .

Under the **User Defined Functions** node, you see a list of functions whose inputs can be constrained.

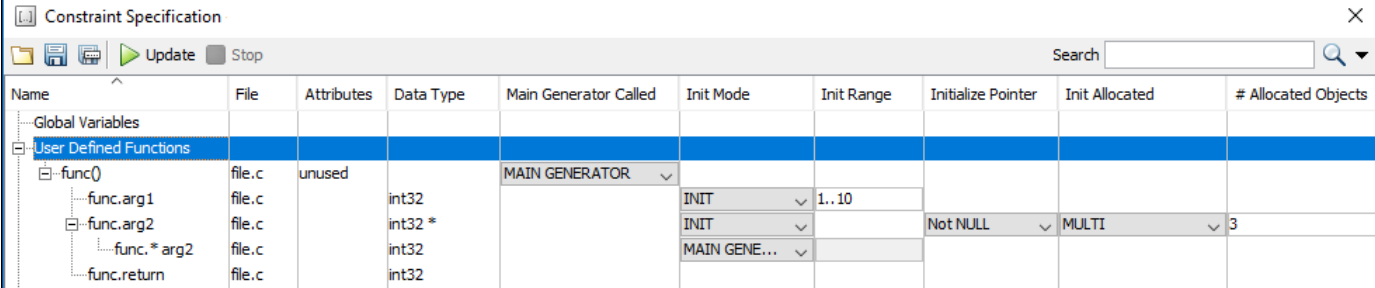
- 3 Expand the node for each function.

You see each function input on a separate row. The inputs have the syntax `function_name.arg1`, `function_name.arg2`, etc.

- 4 Specify your constraints on one or more of the function inputs. For more information, see “External Constraints for Polyspace Analysis” on page 10-7.

For example, in the preceding code:

- To constrain `val` to the range `[1..10]`, select **INIT** for **Init Mode** and enter `1..10` for **Init Range**.
- To specify that `ptr` points to a 3-element array where each element is initialized, select **MULTI** for **Init Allocated** and enter 3 for **# Allocated Objects**.



Name	File	Attributes	Data Type	Main Generator Called	Init Mode	Init Range	Initialize Pointer	Init Allocated	# Allocated Objects
Global Variables									
User Defined Functions									
func()	file.c	unused		MAIN GENERATOR					
func.arg1	file.c		int32		INIT	1..10			
func.arg2	file.c		int32 *		INIT		Not NULL	MULTI	3
func.*arg2	file.c		int32		MAIN GENERATOR				
func.return	file.c		int32						

- Run verification and open the results. On the **Source** pane, place your cursor on the function inputs.

The tooltips display the constraints. For example, in the preceding code, the tooltip displays that `val` has values in `1..10`.

Command Line

Use the option `Constraint setup (-data-range-specifications)` with an XML file specifying your constraint.

For instance, for an analysis with Polyspace Code Prover Server, specify the option as follows:

```
polyspace-code-prover-server -sources filename
                             -data-range-specifications "C:\Polyspace\drs_project1.xml"
```

Create a blank constraint XML template as described in “Specify External Constraints” on page 10-2. In the XML file, locate and constrain the function inputs. The function inputs appear as a scalar or pointer tag in a function tag. The inputs are named as `arg1`, `arg2` and so on. For instance, for the preceding code, the XML structure for the inputs of `func` appear as follows:

```
<function name="func" line="1" attributes="unused"
  main_generator_called="MAIN_GENERATOR" comment="">
  <scalar name="arg1" line="1" base_type="int32"
    complete_type="int32" init_mode="INIT" init_range="1..10" />
  <pointer name="arg2" line="1" complete_type="int32 *"
    init_mode="INIT" initialize_pointer="Not NULL" number_allocated="3"
    init_pointed="MULTI">
    <scalar line="1" base_type="int32" complete_type="int32"
      init_mode="MAIN_GENERATOR" init_range="" />
  </pointer>
  <scalar name="return" line="1" base_type="int32" complete_type="int32"
    init_mode="disabled" init_range="disabled" />
</function>
```

To specify a constraint on a function input, set the attributes `init_mode` and `init_range` for scalar variables, and `init_pointed` and `number_allocated` for pointer variables.

- To constrain `val` to the range `[1..10]`, set the `init_mode` attribute of the tag with name `arg1` to `INIT` and `init_range` to `1..10`.

- To specify that `ptr` points to a 3-element array where each element is initialized, set the `init_mode` attribute of the tag with name `arg2` to `INIT`, `init_pointed` to `MULTI` and `number_allocated` to 3.

See Also

Constraint setup (-data-range-specifications)

More About

- “Specify External Constraints” on page 10-2
- “External Constraints for Polyspace Analysis” on page 10-7
- “Constrain Global Variable Range” on page 10-13

XML File Format for Constraints

For a more precise Polyspace analysis, you can specify constraints on global variables, function inputs and stubbed functions. You can specify the constraints in the user interface of the Polyspace desktop products or at the command line as an XML file. For the general workflow, see “Specify External Constraints” on page 10-2.

This topic describes details of the constraint XML file schema. You typically require this information only if you create a constraint XML from scratch. If you run a verification once, the software automatically generates a template constraint file `drs-template.xml` in your results folder. Instead of creating a constraint XML file from scratch, it is easier to edit this template XML file to specify your constraints. For some examples, see:

- “Constrain Global Variable Range” on page 10-13
- “Constrain Function Inputs” on page 10-16

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

You can also see the information in this topic and the underlying XML schema in `polyspaceroot\polyspace\drs`. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

Syntax Description — XML Elements

The constraints file contains the following XML elements:

- `<global>` element — Declares the global scope, and is the root element of the XML file.
- `<file>` element — Declares a file scope. Must be enclosed in the `<global>` element. May enclose any variable or function declaration. Static variables must be enclosed in a file element to avoid conflicts.
- `<scalar>` element— Declares an integer or a floating point variable. May be enclosed in any recognized element, but cannot enclose any element. Sets `init/permanent/global` asserts on variables.
- `<pointer>` element — Declares a pointer variable. May enclose any other variable declarations (including itself), to define the pointed objects. Specifies what value is written into pointer (NULL or not), how many objects are allocated and how the pointed objects are initialized.
- `<array>` element — Declares an array variable. May enclose any other variable definition (including itself), to define the members of the array.
- `<struct>` element — Declares a structure variable or object (instance of class). May enclose any other variable definition (including itself), to define the fields of the structure.
- `<function>` element — Declares a function or class method scope. May enclose any variable definition, to define the arguments and the return value of the function. Arguments should be named `arg1`, `arg2`, ..., `argn` and the return value should be called `return`.

The following notes apply to specific fields in each XML element:

- **(*)** — Fields used only by the GUI. These fields are not mandatory for verification to accept the ranges. The field `line` contains the line number where the variable is declared in the source code, `complete_type` contains a string with the complete variable type, and `base_type` is used by the

GUI to compute the min and max values. The field comment is used to add information about any node.

- **(**)** — The field name is mandatory for scope elements `<file>` and `<function>` (except for function pointers). For other elements, the name must be specified when declaring a root symbol or a struct field.
- **(***)** — If more than one attribute applies to the variable, the attributes must be separated by a space. Only the static attribute is mandatory, to avoid conflicts between static variables having the same name. An attribute can be defined multiple times without impact.
- **(****)** — This element is used only by the GUI, to determine which `init` modes are allowed for the current element (according to its type). The value works as a mask, where the following values are added to specify which modes are allowed:
 - **1**: The mode “NO” is allowed.
 - **2** : The mode “INIT” is allowed.
 - **4**: The mode “PERMANENT” is allowed.
 - **8**: The mode “MAIN_GENERATOR” is allowed.

For example, the value “**10**” means that modes “INIT” and “MAIN_GENERATOR” are allowed. To see how this value is computed, refer to “Valid Modes and Default Values” on page 10-23.

- **(*****)** — A sub-element of a pointer (i.e. a pointed object) will be taken into account only if `init_pointed` is equal to `SINGLE`, `MULTI`, `SINGLE_CERTAIN_WRITE` or `MULTI_CERTAIN_WRITE`.
- **(*****)** — `SINGLE_CERTAIN_WRITE` or `MULTI_CERTAIN_WRITE` are available for parameters and return values of stubbed functions only if they are pointers. If the parameter or return value is a structure and the structure has a pointer field, they are also available for the pointer field.

<file> Element

Field	Syntax
name	<i>filepath_or_filename</i>
comment	<i>string</i>

<scalar> Element

Field	Syntax
name (**)	<i>name</i>
line (*)	<i>line</i>
base_type (*)	intx uintx floatx
Attributes (***)	volatile extern static const
complete_type (*)	<i>type</i>

Field	Syntax
init_mode	MAIN_GENERATOR IGNORE INIT PERMANENT disabled unsupported
init_modes_allowed (*)	<i>single value (****)</i>
init_range	<i>range</i> disabled unsupported
global_assert	YES NO disabled unsupported
assert_range	<i>range</i> disabled unsupported
comment(*)	<i>string</i>

<pointer> Element

Field	Syntax
Name (**)	<i>name</i>
line (*)	<i>line</i>
Attributes (***)	volatile extern static const
complete_type (*)	<i>type</i>
init_mode	MAIN_GENERATOR IGNORE INIT PERMANENT disabled unsupported
init_modes_allowed (*)	<i>single value (****)</i>
initialize_pointer	May be: NULL Not NULL NULL
number_allocated	<i>single value</i> disabled unsupported

Field	Syntax
init_pointed (*****)	MAIN_GENERATOR NONE SINGLE MULTI SINGLE_CERTAIN_WRITE MULTI_CERTAIN_WRITE disabled
comment	<i>string</i>

<array> and <struct> Elements

Field	Syntax
Name (**)	<i>name</i>
line (*)	<i>line</i>
complete_type (*)	<i>type</i>
attributes (***)	volatile extern static const
comment	<i>string</i>

<function> Element

Field	Syntax
Name (**)	<i>name</i>
line (*)	<i>line</i>
main_generator_called	MAIN_GENERATOR YES NO disabled
attributes (***)	static extern unused
comment	<i>string</i>

Valid Modes and Default Values

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
Global variables	Base type	Unqualified/static/const scalar	MAIN_GENERATOR IGNORE INIT PERMANENT	YES NO			Main generator dependent
		Volatile scalar	PERMANENT	disabled			PERMANENT min..max
		Extern scalar	INIT PERMANENT	YES NO			INIT min..max
	Struct	Struct field	Refer to field type				
	Array	Array element	Refer to element type				
Global variables	Pointer	Unqualified/static/const scalar	MAIN_GENERATOR IGNORE INIT		May be NULL Not NULL NULL	NONE SINGLE MULTI	Main generator dependent
		Volatile pointer	un-supported		un-supported	un-supported	
		Extern pointer	IGNORE INIT		May be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May be NULL max MULTI
		Pointed volatile scalar	un-supported	un-supported			
		Pointed extern scalar	INIT	un-supported			INIT min..max
		Pointed other scalars	MAIN_GENERATOR INIT	un-supported			MAIN_GENERATOR dependent
		Pointed pointer	MAIN_GENERATOR INIT/	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	MAIN_GENERATOR dependent
		Pointed function	un-supported	un-supported			
Function parameters	Userdef function	Scalar parameters	MAIN_GENERATOR INIT	un-supported			INIT min..max

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default	
		Pointer parameters	MAIN_GENERATOR INIT	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May be NULL max MULTI	
		Other parameters	Refer to parameter type					
	Stubbed function	Scalar parameter	disabled		un-supported			
		Pointer parameters	disabled			disabled	NONE SINGLE MULTI SINGLE_CERTAIN_ WRITE MULTI_CERTAIN_ WRITE	MULTI
		Pointed parameters	PERMANENT		un-supported			PERMANENT min..max
		Pointed const parameters	disabled		un-supported			
Function return	Userdef function	Return	disabled	un-supported	disabled	disabled		
	Stubbed function	Scalar return	PERMANENT	un-supported			PERMANENT min..max	
		Pointer return	PERMANENT		un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI SINGLE_CERTAIN_ WRITE MULTI_CERTAIN_ WRITE	PERMANENT May be NULL max MULTI

See Also

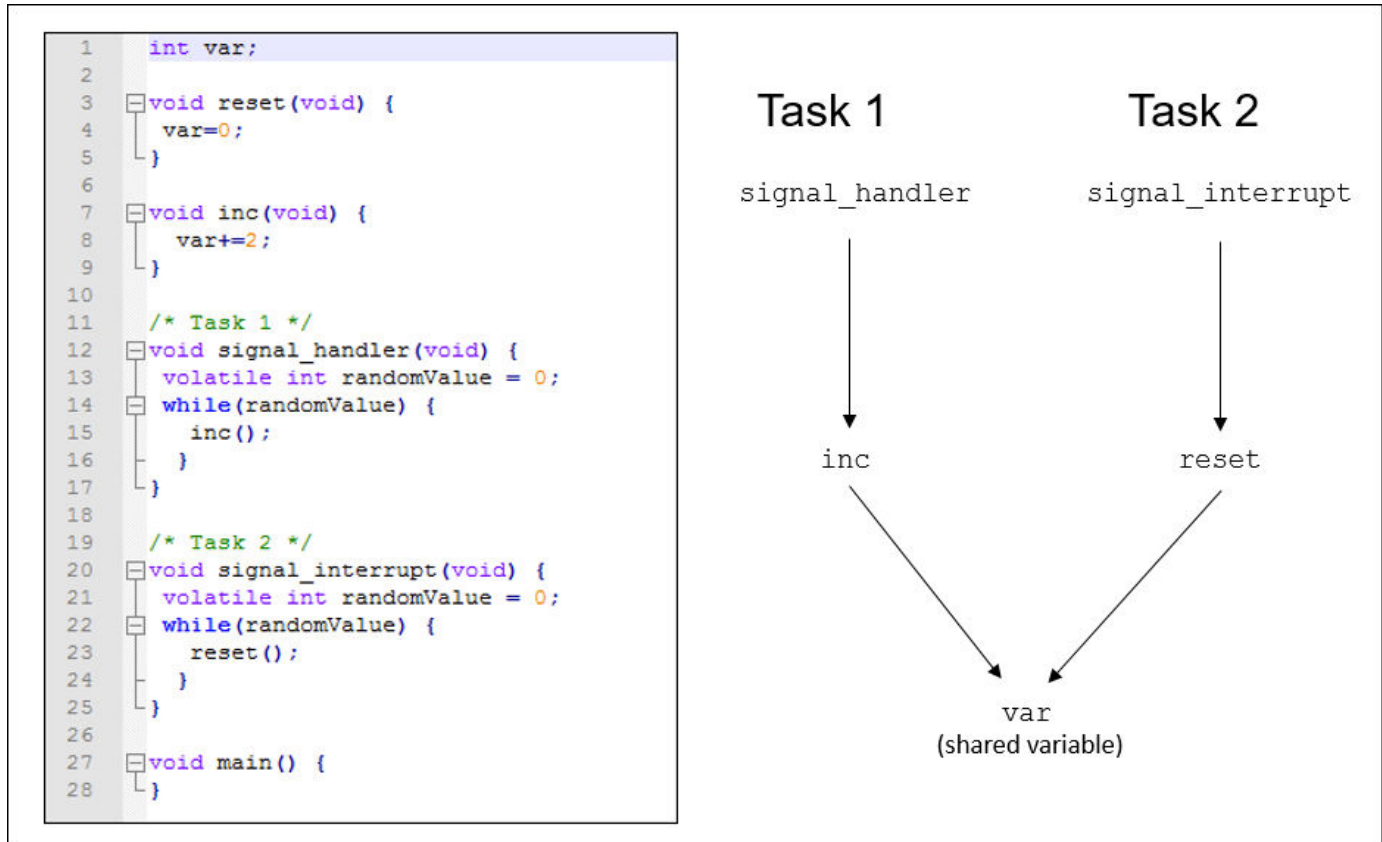
More About

- “Specify External Constraints” on page 10-2
- “Constrain Global Variable Range” on page 10-13
- “Constrain Function Inputs” on page 10-16

Configure Multitasking Analysis

Analyze Multitasking Programs in Polyspace

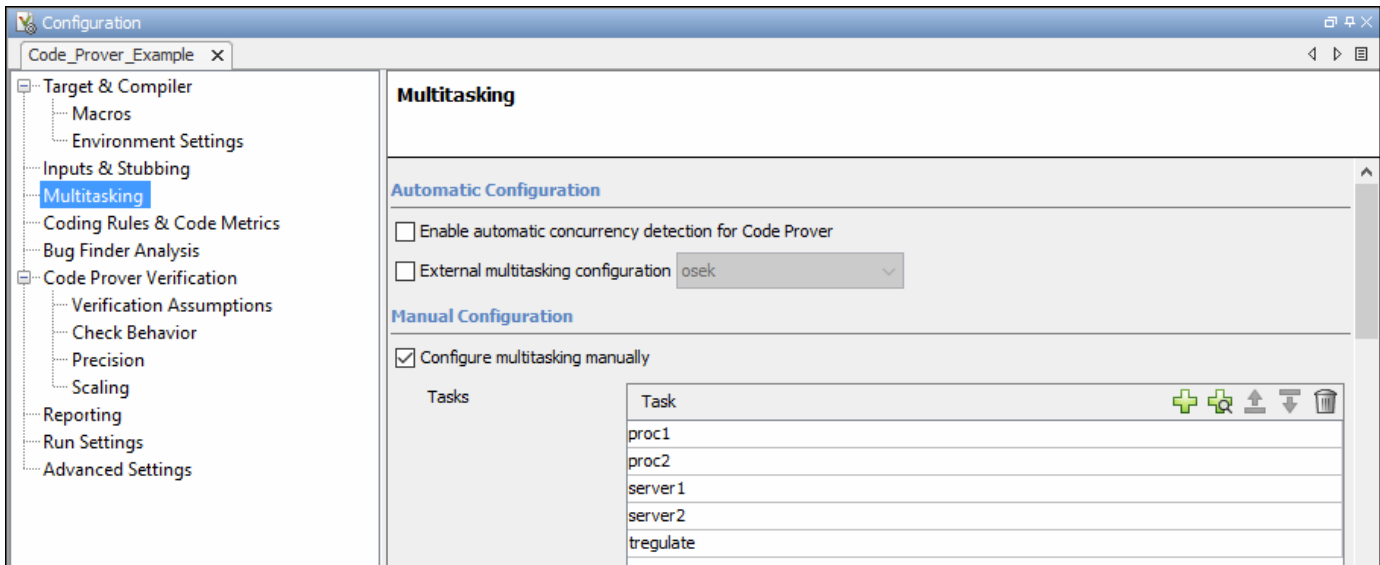
With Polyspace, you can analyze programs where multiple threads (tasks) run concurrently.



In addition to regular run-time checks, the analysis looks for issues specific to concurrent execution:

- Data races, deadlocks, consecutive or missing locks and unlocks (Bug Finder)
- Unprotected shared variables (Code Prover)

Configure Analysis



If your code uses multitasking primitives from certain families, for instance, `pthread_create` for thread creation:

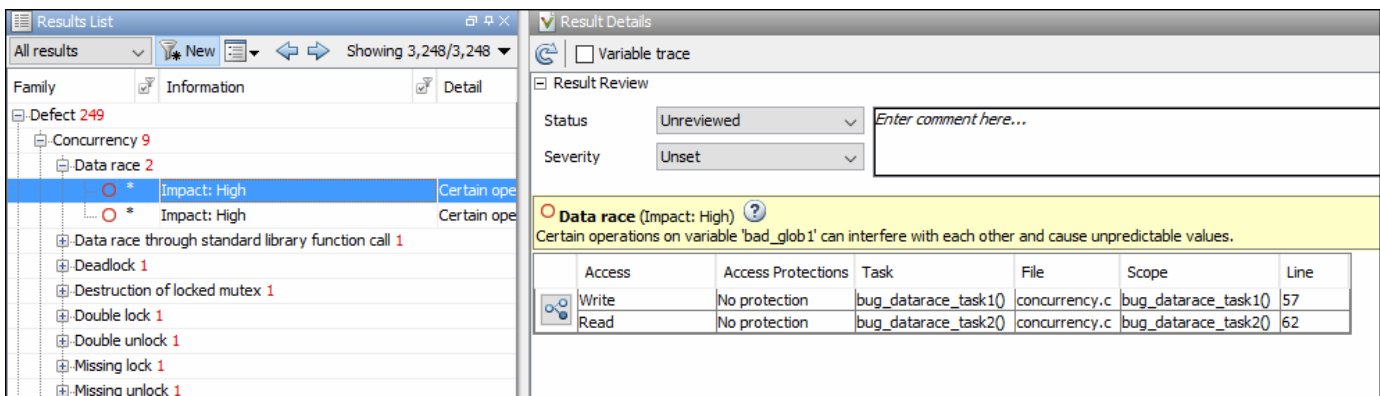
- In Bug Finder, the analysis detects them and extracts your multitasking model from the code.
- In Code Prover, you must enable this automatic detection explicitly.

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

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

Review Analysis Results

Bug Finder



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

Code Prover


The screenshot displays the Polyspace Code Prover interface. On the left, the 'Results List' pane shows a tree view of analysis results. Under 'Global Variable', the 'Potentially unprotected variable' category is expanded, listing several variables: PowerLevel, SHR4, SHR2, SHR, and SHR5. The 'PowerLevel' variable is selected, showing it is shared across tasks.

The 'Result Details' pane on the right provides a detailed view of the selected error. It includes a 'Result Review' section with 'Status' set to 'Unreviewed' and 'Severity' set to 'Unset'. Below this, a yellow warning box states: 'Potentially unprotected variable' and 'Variable 'tasks1.PowerLevel' is shared among several tasks. Some operations on variable 'tasks1.PowerLevel' have no common protection. Read by task: server1 server2 tregulate. Written by task: server1 server2 tregulate.'

At the bottom of the 'Result Details' pane, a table lists the conflicting operations:

Event	File	Scope	Line
Written value: -10000	main.c	main()	36
Written value: 0	tasks1.c	_init_globals()	26
Written value: [-2147483639 .. 2 ³¹ -1]	tasks2.c	Increase_PowerLevel()	19
Read value: [-2147483640 .. 2 ³¹ -1]	tasks1.c	orderregulate()	40
Read value: [-2147483640 .. 2 ³¹ -1]	tasks2.c	Increase_PowerLevel()	19
Read value: [-2147483640 .. 2 ³¹ -1]	tasks2.c	Compute_Injection()	34
Read value: [-2147483640 .. 2 ³¹ -1]	tasks2.c	Get_PowerLevel()	41

The Code Prover analysis exhaustively checks if shared global variables are protected from concurrent access. 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.

See Also

More About

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

Auto-Detection of Thread Creation and Critical Section in Polyspace

With Polyspace, you can analyze programs where multiple threads run concurrently. Polyspace can analyze your multitasking code for data races, deadlocks and other concurrency defects, if the analysis is aware of the concurrency model in your code. In some situations, Polyspace can detect thread creation and critical sections in your code automatically. Bug Finder detects them by default. In Code Prover, you enable automatic detection using the option `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.

For the multitasking code analysis workflow, see “Analyze Multitasking Programs in Polyspace” on page 11-2.

If your thread creation function is not detected automatically:

- You can also map the function to a thread-creation function that Polyspace can detect automatically. Use the option `-code-behavior-specifications`.
- Otherwise, you must manually model your multitasking threads by using configuration options. See “Configuring Polyspace Multitasking Analysis Manually” on page 11-16.

Multitasking Routines that Polyspace Can Detect

Polyspace can detect thread creation and critical sections if you use primitives from these groups. Polyspace recognizes calls to these routines as the creation of a new thread or as the beginning or end of a critical section.

POSIX

Thread creation: `pthread_create`

Critical section begins: `pthread_mutex_lock`

Critical section ends: `pthread_mutex_unlock`

VxWorks

Thread creation: `taskSpawn`

Critical section begins: `semTake`

Critical section ends: `semGive`

To activate automatic detection of concurrency primitives for VxWorks®, in the user interface of the Polyspace desktop products, use the `VxWorks` template. For more information on templates, see “Create Project Using Configuration Template” on page 1-15. At the command-line, use these options:

```
-D1=CPU=I80386
-D2=__GNUC__=2
-D3=__OS_VXWORKS
```

Concurrency detection is possible only if the multitasking functions are created from an entry point named `main`. If the entry point has a different name, such as `vxworks_entry_point`, do one of the following:

- Provide a `main` function.
- Preprocessor definitions (-D): In preprocessor definitions, set `vxworks_entry_point=main`.

Windows

Thread creation: `CreateThread`

Critical section begins: `EnterCriticalSection`

Critical section ends: `LeaveCriticalSection`

µC/OS II

Thread creation: `OSTaskCreate`

Critical section begins: `OSMutexPend`

Critical section ends: `OSMutexPost`

C++11

Thread creation: `std::thread::thread`

Critical section begins: `std::mutex::lock`

Critical section ends: `std::mutex::unlock`

For autodetection of C++11 threads, explicitly specify paths to your compiler header files or use `polyspace-configure`.

For instance, if you use `std::thread` for thread creation, explicitly specify the path to the folder containing `thread.h`.

See also “Limitations of Automatic Thread Detection” on page 11-11.

C11

Thread creation: `thr_create`

Critical section begins: `mtx_lock`

Critical section ends: `mtx_unlock`

Example of Automatic Thread Detection

The following multitasking code models five philosophers sharing five forks. The example uses POSIX[®] thread creation routines and illustrates a classic example of a deadlock. Run Bug Finder on this code to see the deadlock.

```
#include "pthread.h"
#include <stdio.h>
#include <unistd.h>

pthread_mutex_t forks[5];

void* philo1(void* args)
{
    while (1) {
        printf("Philosopher 1 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[0]);
        printf("Philosopher 1 takes left fork\n");
        pthread_mutex_lock(&forks[1]);
        printf("Philosopher 1 takes right fork\n");
        printf("Philosopher 1 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[1]);
        printf("Philosopher 1 puts down right fork\n");
        pthread_mutex_unlock(&forks[0]);
        printf("Philosopher 1 puts down left fork\n");
    }
    return NULL;
}

void* philo2(void* args)
{
    while (1) {
        printf("Philosopher 2 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[1]);
        printf("Philosopher 2 takes left fork\n");
        pthread_mutex_lock(&forks[2]);
        printf("Philosopher 2 takes right fork\n");
        printf("Philosopher 2 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[2]);
        printf("Philosopher 2 puts down right fork\n");
        pthread_mutex_unlock(&forks[1]);
        printf("Philosopher 2 puts down left fork\n");
    }
    return NULL;
}

void* philo3(void* args)
{
    while (1) {
        printf("Philosopher 3 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[2]);
        printf("Philosopher 3 takes left fork\n");
        pthread_mutex_lock(&forks[3]);
        printf("Philosopher 3 takes right fork\n");
        printf("Philosopher 3 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[3]);
        printf("Philosopher 3 puts down right fork\n");
    }
}
```



```
        pthread_mutex_unlock(&forks[2]);
        printf("Philosopher 3 puts down left fork\n");
    }
    return NULL;
}

void* philo4(void* args)
{
    while (1) {
        printf("Philosopher 4 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[3]);
        printf("Philosopher 4 takes left fork\n");
        pthread_mutex_lock(&forks[4]);
        printf("Philosopher 4 takes right fork\n");
        printf("Philosopher 4 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[4]);
        printf("Philosopher 4 puts down right fork\n");
        pthread_mutex_unlock(&forks[3]);
        printf("Philosopher 4 puts down left fork\n");
    }
    return NULL;
}

void* philo5(void* args)
{
    while (1) {
        printf("Philosopher 5 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[4]);
        printf("Philosopher 5 takes left fork\n");
        pthread_mutex_lock(&forks[0]);
        printf("Philosopher 5 takes right fork\n");
        printf("Philosopher 5 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[0]);
        printf("Philosopher 5 puts down right fork\n");
        pthread_mutex_unlock(&forks[4]);
        printf("Philosopher 5 puts down left fork\n");
    }
    return NULL;
}

int main(void)
{
    pthread_t ph[5];
    pthread_create(&ph[0], NULL, philo1, NULL);
    pthread_create(&ph[1], NULL, philo2, NULL);
    pthread_create(&ph[2], NULL, philo3, NULL);
    pthread_create(&ph[3], NULL, philo4, NULL);
    pthread_create(&ph[4], NULL, philo5, NULL);

    pthread_join(ph[0], NULL);
    pthread_join(ph[1], NULL);
    pthread_join(ph[2], NULL);
    pthread_join(ph[3], NULL);
    pthread_join(ph[4], NULL);
}
```

```

return 1;
}

```

Each philosopher needs two forks to eat, a right and a left fork. The functions `philo1`, `philo2`, `philo3`, `philo4`, and `philo5` represent the philosophers. Each function requires two `pthread_mutex_t` resources, representing the two forks required to eat. All five functions run at the same time in five concurrent threads.

However, a deadlock occurs in this example. When each philosopher picks up their first fork (each thread locks one `pthread_mutex_t` resource), all the forks are being used. So, the philosophers (threads) wait for their second fork (second `pthread_mutex_t` resource) to become available. However, all the forks (resources) are being held by the waiting philosophers (threads), causing a deadlock.

Naming Convention for Automatically Detected Threads

If you use a function such as `pthread_create()` to create new threads (tasks), each thread is associated with a unique identifier. For instance, in this example, two threads are created with identifiers `id1` and `id2`.

```

pthread_t* id1, id2;

void main()
{
    pthread_create(id1, NULL, start_routine, NULL);
    pthread_create(id2, NULL, start_routine, NULL);
}

```

If a data race occurs between the threads, the analysis can detect it. When displaying the results, the threads are indicated as `task_id`, where `id` is the identifier associated with the thread. In the preceding example, the threads are identified as `task_id1` and `task_id2`.

If a thread identifier is:

- Local to a function, the thread name shows the function.

For instance, the thread created below appears as `task_f:id`

```

void f(void)
{
    pthread_t* id;
    pthread_create(id, NULL, start_routine, NULL);
}

```

- A field of a structure, the thread name shows the structure.

For instance, the thread created below appears as `task_a#id`

```

struct {pthread_t* id; int x;} a;
pthread_create(a.id, NULL, start_routine, NULL);

```

- An array member, the thread name shows the array.

For instance, the thread created below appears as `task_tab[1]`.

```
pthread_t* tab[10];
pthread_create(tab[1],NULL,start_routine,NULL);
```

If you create two threads with distinct thread identifiers, but you use the same local variable name for the thread identifiers, the name of the second thread is modified to distinguish it from the first thread. For instance, the threads below appear as `task_func:id` and `task_func:id:1`.

```
void func()
{
    {
        pthread_t id;
        pthread_create(&id, NULL, &task, NULL);
    }
    {
        pthread_t id;
        pthread_create(&id, NULL, &task, NULL);
    }
}
```

Limitations of Automatic Thread Detection

The multitasking model extracted by Polyspace does not include some features. Polyspace cannot model:

- Thread priorities and attributes — Ignored by Polyspace.
- Recursive semaphores.
- Unbounded thread identifiers, such as `extern pthread_t ids[]` — Warning.
- Calls to concurrency primitive through high-order calls — Warning.
- Aliases on thread identifiers — Polyspace over-approximates when the alias is used.
- Termination of threads — Polyspace ignores `pthread_join` and `thrd_join`. Polyspace replaces `pthread_exit` and `thrd_exit` by a standard `exit`.
- (Polyspace Bug Finder only) Creation of multiple threads through multiple calls to the same function with different pointer arguments.

Example

In this example, Polyspace considers that only one thread is created.

```
pthread_t id1, id2;
void start(pthread_t* id)
{
    pthread_create(id, NULL, start_routine, NULL);
}
void main()
{
    start(&id1);
    start(&id2);
}
```

- (Polyspace Code Prover only) Shared local variables — Only global variables are considered shared. If a local variable is accessed by multiple threads, the analysis does not take into account the shared nature of the variable.

Example

In this example, the analysis does not take into account that the local variable `x` can be accessed by both `task1` and `task2` (after the new thread is created).

```
#include <pthread.h>
#include <stdlib.h>

void* task2(void* args)
{
    int* x = (int*) args;
    *x = 1;
    return (void*)x;
}

void task1()
{
    int x;
    x = 2;
    pthread_t id;
    (void)pthread_create(&id, NULL, task2, (void*) &x);
    /* x (local var) passed to task2 */
    x = 3 ;

    /* Unknown thread priority means x = 1 OR x = 3.*/
    /* However, the analysis considers x = 3 */
    /* Assertion below is green */
    assert(x == 3);
}

int main(void)
{
    task1();
    return 0;
}
```

- (Polyspace Code Prover only) Shared dynamic memory — Only global variables are considered shared. If a dynamically allocated memory region is accessed by multiple threads, the analysis does not take into account its shared nature.

Example

In this example, the analysis does not take into account that `lx` points to a shared memory region. The region can be accessed by both `task1` and `task2` (after the new thread is created). The Code Prover analysis also reports `lx` as a non-shared variable.

```

#include <pthread.h>
#include <stdlib.h>

static int* lx;

void* task2(void* args)
{
    int* x = (int*) args;
    *x = 1;
    return (void*)x;
}

void task1()
{
    pthread_t id;
    lx = (int*)malloc(sizeof(int));

    if (lx == NULL) exit(1);

    (void)pthread_create(&id, NULL, task2, (void*) lx);

    *lx = 3 ;

    /* Unknown thread priority means *lx = 1 OR *lx = 3.*/
    /* However, the analysis considers *lx = 3 */
    /* Assertion below is green */
    assert(*lx == 3);
}

int main(void)
{
    task1();
    return 0;
}

```

- Number of tasks created with CreateThread when threadId is set to NULL— When you create multiple threads that execute the same function, if the last argument of CreateThread is NULL, Polyspace only detects one instance of this function, or task.

Example

In this example, Polyspace detects only one instance of thread_function1(), but 10 instances of thread_function2().

```

#include <windows.h>

#define MAX_LOOP_THREADS 10

DWORD WINAPI thread_function1(LPVOID data) {}
DWORD WINAPI thread_function2(LPVOID data) {}

HANDLE hds1[MAX_LOOP_THREADS];
HANDLE hds2[MAX_LOOP_THREADS];
DWORD threadId[MAX_LOOP_THREADS];

int main(void)
{
    for (int i = 0; i < MAX_LOOP_THREADS; i++) {
        hds1[i] = CreateThread(NULL, 0, thread_function1, NULL, 0, NULL);
        hds2[i] = CreateThread(NULL, 0, thread_function2, NULL, 0, &threadId[i]);
    }

    return 0;
}

```

- (C++11 only) If you use lambda expressions as start functions during thread creation, Polyspace does not detect shared variables in the lambda expressions.

Example

In this example, Polyspace does not detect that the variable `y` used in the lambda expressions is shared between two threads. As a result, Bug Finder, for instance, does not show a **Data race** defect.

```

#include <thread>
int y;
int main() {
    std::thread t1([] {y++;});
    std::thread t2([] {y++;});
    t1.join();
    t2.join();
    return 0;
}

```

- (C++11 threads with Polyspace Code Prover only) String literals as thread function argument — Code Prover shows a red **Illegally dereferenced pointer** error if the thread function has an `std::string&` parameter and you pass a string literal argument.

Example

In this example, the thread function `foo` has an `std::string&` parameter. When starting a thread, a string literal is passed as argument to this function, which undergoes an implicit conversion to `std::string` type. Code Prover loses track of the original string literal in this conversion. Therefore, a dashed red underline appears on `operator<<` in the body of `foo` and a red **Illegally dereferenced pointer** check in the body of `operator<<`.

```
#include <iostream>
#include <thread>

using namespace std;

void foo(const std::string& f) {
    std::cout << f;
}

void main() {
    std::thread t1(foo, "foo_arg");
}
```

To work around this issue, assign the string literal to a temporary variable and pass the variable as argument to the thread function.

```
#include <iostream>
#include <thread>

using namespace std;

void foo(const std::string& f) {
    std::cout << f;
}

void main() {
    std::string str = "foo_arg";
    std::thread t1(foo, str);
}
```

See Also

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

More About

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

Configuring Polyspace Multitasking Analysis Manually

With Polyspace, you can analyze programs where multiple threads run concurrently. In some situations, Polyspace can detect thread creation and critical sections in your code automatically. See “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 11-5.

If your code has functions that are intended for concurrent execution, but that cannot be detected automatically, you must specify them before analysis. If these functions operate on a common variable, you must also specify protection mechanisms for those operations.

For the multitasking code analysis workflow, see “Analyze Multitasking Programs in Polyspace” on page 11-2.

Specify Options for Multitasking Analysis

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

- **Entry points (-entry-points)**: Specify noncyclic entry point functions.
Do not specify `main`. Polyspace implicitly considers `main` as an entry point function.
- **Cyclic tasks (-cyclic-tasks)**: Specify functions that are scheduled at periodic intervals.
- **Interrupts (-interrupts)**: Specify functions that can run asynchronously.
- **Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)**: Specify functions that disable and reenables interrupts (Bug Finder only).
- **Critical section details (-critical-section-begin -critical-section-end)**: Specify functions that begin and end critical sections.
- **Temporally exclusive tasks (-temporal-exclusions-file)**: Specify groups of functions that are temporally exclusive.
- **-preemptable-interrupts**: Specify functions that have lower priority than interrupts, but higher priority than tasks (preemptable or non-preemptable).
Only the Bug Finder analysis considers priorities.
- **-non-preemptable-tasks**: Specify functions that have higher priority than tasks, but lower priority than interrupts (preemptable or non-preemptable).
Only the Bug Finder analysis considers priorities.

Adapt Code for Code Prover Multitasking Analysis

The multitasking analysis in Code Prover is more exhaustive about finding potentially unprotected shared variables and therefore follows a strict model.

Tasks and interrupts must be void-void functions.

Functions that you specify as tasks and interrupts must have the prototype:

```
void func(void);
```


Suppose you want to specify a function `func` that takes `int` arguments:

```
void 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;
    func(arg);
}
```

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 22-72.

Suppose you want to specify the `main` function as a cyclic task.

```
void performTask1Cycle(void);
void performTask2Cycle(void);
```

```
void main() {
    while(1) {
        performTask1Cycle();
    }
}
```

```
void task2() {
    while(1) {
        performTask2Cycle();
    }
}
```

Replace the definition of `main` with:

```
#ifdef POLYSPACE
void main() {
}
void task1() {
    while(1) {
        performTask1Cycle();
    }
}
#else
void main() {
    while(1) {
        performTask1Cycle();
    }
}
#endif
```

The replacement defines an empty `main` and places the content of `main` into another function `task1` if a macro `POLYSPACE` is defined. Define the macro `POLYSPACE` using the option `Preprocessor definitions (-D)` and specify `task1` for the option `Tasks (-entry-points)`.

This assumption does not apply to automatically detected threads. For instance, a `main` function can create threads using `pthread_create`.

All tasks and interrupts can interrupt each other.

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

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

The Polyspace multitasking analysis assumes that a task or interrupt cannot interrupt itself.

All tasks and interrupts can run any number of times in any sequence.

The Code Prover analysis considers that all tasks and interrupts can run any number of times in any sequence.

Suppose in this example, you specify `reset` and `inc` as cyclic tasks. The analysis shows an overflow on the operation `var+=2`.

```
void reset(void) {
    var=0;
}

void inc(void) {
    var+=2;
}
```

Suppose you want to model a scheduling of tasks such that `reset` executes after `inc` has executed five times. Write a wrapper function that implements this sequence. Specify this new function as a cyclic task instead of `reset` and `inc`.

```
void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        inc();
        inc();
        inc();
        inc();
        inc();
        reset();
    }
}
```

Suppose you want to model a scheduling of tasks such that `reset` executes after `inc` has executed zero to five times. Write a wrapper function that implements this sequence. Specify this new function as a cyclic task instead of `reset` and `inc`.

```
void task() {
    volatile int randomValue = 0;
```

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

See Also

More About

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

Protections for Shared Variables in Multitasking Code

If your code is intended for multitasking, tasks in your code can access a common shared variable. To prevent data races, you can protect read and write operations on the variable. This topic shows the various protection mechanisms that Polyspace can recognize.

Detect Unprotected Access

Access	Access Protections	Task	File	Scope	Line
Write	No protection	bug_datarace_task1()	concurrency.c	bug_datarace_task1()	57
Read	No protection	bug_datarace_task2()	concurrency.c	bug_datarace_task2()	62

You can detect an unprotected access using either Bug Finder or Code Prover. Code Prover is more exhaustive and proves if a shared variable is protected from concurrent access.

- Bug Finder detects an unprotected access using the result **Data race**. See [Data race](#).
- Code Prover detects an unprotected access using the result **Shared unprotected global variable**. See [Potentially unprotected variable](#).

Suppose you analyze this code, specifying `signal_handler_1` and `signal_handler_2` as cyclic tasks. Use the analysis option `Cyclic tasks (-cyclic-tasks)`.

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void signal_handler_1(void) {
    reset();
    inc();
    inc();
}

void signal_handler_2(void) {
    shared_var = INT_MAX;
}
```

```

}

void main() {
}

```

Bug Finder shows a data race on `shared_var`. Code Prover shows that `shared_var` is a potentially unprotected shared variable. Code Prover also shows that the operation `shared_var += 2` can overflow. The overflow occurs if the call to `inc` in `signal_handler_1` immediately follows the operation `shared_var = INT_MAX` in `signal_handler_2`.

Protect Using Critical Sections

One possible solution is to protect operations on shared variables using critical sections.

In the preceding example, modify your code so that operations on `shared_var` are in the same critical section. Use the functions `take_semaphore` and `give_semaphore` to begin and end the critical sections. To specify these functions that begin and end critical sections, use the analysis options `Critical section details (-critical-section-begin -critical-section-end)`.

```

#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

/* Declare lock and unlock functions */
void take_semaphore(void);
void give_semaphore(void);

void signal_handler_1() {
    /* Begin critical section */
    take_semaphore();
    reset();
    inc();
    inc();
    /* End critical section */
    give_semaphore();
}

void signal_handler_2() {
    /* Begin critical section */
    take_semaphore();
    shared_var = INT_MAX;
    /* End critical section */
    give_semaphore();
}

```

```
void main() {  
}
```

You do not see the data race in Bug Finder. Code Prover proves that the shared variable is protected. You also do not see the overflow because the call to `reset()` in `signal_handler_1` always precedes calls to `inc()`.

You can also use primitives such as the POSIX functions `pthread_mutex_lock` and `pthread_mutex_unlock` to begin and end critical sections. For a list of primitives that Polyspace can detect automatically, see “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 11-5.

Protect Using Temporally Exclusive Tasks

Another possible solution is to specify a group of tasks as temporally exclusive. Temporally exclusive tasks cannot interrupt each other.

In the preceding example, specify that `signal_handler_1` and `signal_handler_2` are temporally exclusive. Use the option `Temporally exclusive tasks (-temporal-exclusions-file)`.

You do not see the data race in Bug Finder. Code Prover proves that the shared variable is protected. You also do not see the overflow because the call to `reset()` in `signal_handler_1` always precedes calls to `inc()`.

Protect Using Priorities

Another possible solution is to specify that one task has higher priority over another.

In the preceding example, specify that `signal_handler_1` is an interrupt. Retain `signal_handler_2` as a cyclic task. Use the options `Cyclic tasks (-cyclic-tasks)` and `Interrupts (-interrupts)`.

Bug Finder does not show the data race defect anymore. The reason is this:

- The operation `shared_var = INT_MAX` in `signal_handler_2` is atomic. Therefore, the operations in `signal_handler_1` cannot interrupt it.
- The operations in `signal_handler_1` cannot be interrupted by the operation in `signal_handler_2` because `signal_handler_1` has higher priority.

You can specify up to four different priorities with these options (with highest priority listed first):

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

A task with higher priority is atomic with respect to a task with lower priority. Note that the checker `Data race including atomic operations` ignores the difference in priorities and continues to

show the data race. See also “Define Preemptable Interrupts and Nonpreemptable Tasks” (Polyspace Bug Finder).

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

Protect By Disabling Interrupts

In a Bug Finder analysis, you can protect a group of operations by disabling all interrupts. Use the option `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

After you call a routine to disable interrupts, all subsequent operations are atomic until you call another routine to reenale interrupts. The operations are atomic with respect to operations in all other tasks.

See Also

More About

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

Define Atomic Operations in Multitasking Code

In code with multiple threads, you can use Polyspace Bug Finder to detect data races or Polyspace Code Prover to list potentially unprotected shared variables.

To determine if a variable shared between multiple threads is protected against concurrent access, Polyspace checks if the operations on the variable are atomic.

Nonatomic Operations

If an operation is nonatomic, Polyspace considers that the operation involves multiple steps. These steps do not need to occur together and can be interrupted by operations in other threads.

For instance, consider these two operations in two different threads:

- Thread 1: `var++;`

This operation is nonatomic because it takes place in three steps: reading `var`, incrementing `var`, and writing back `var`.

- Thread 2: `var = 0;`

This operation is atomic if the size of `var` is less than the word size on the target. See details below for how Polyspace determines the word size.

If the two operations are not protected (by using, for instance, critical sections), the operation in the second thread can interrupt the operation in the first thread. If the interruption happens after `var` is incremented in the first thread but before the incremented value is written back, you can see unexpected results.

What Polyspace Considers as Nonatomic

Code Prover considers all operations as nonatomic unless you protect them, for instance, by using critical sections. See “Define Specific Operations as Atomic” on page 11-25.

Bug Finder considers an operation as nonatomic if it can translate into more than one machine instruction. For instance:

- The operation can involve both a read and write operation. For example, `var++` involves reading the value of `var`, increasing the value by one and writing the increased value back to `var`.
- The operation can involve a 64-bit variable on a 32-bit target. For example, the operation

```
long long var1, var2;  
var1=var2;
```

involves two steps in copying the content of `var2` to `var1` on certain targets.

Polyspace uses the **Pointer** size for your **Target processor type** as the threshold to compute atomicity. For instance, if you use `i386` as your **Target processor type**, the **Pointer** size is 32 bits and **Long long** and **Double** sizes are both 64 bits. Therefore, Polyspace considers copying one `long long` or `double` variable to another as nonatomic.

See also `Target processor type (-target)`.

- The operation can involve writing the return value of a function call to a shared variable. For example, the operation `x=func()` involves calling `func` and writing the return value of `func` to `x`.

To detect data races where at least one of the two interrupting operations is nonatomic, enable the Bug Finder checker `Data race`. To remove this constraint on the checker, enable `Data race including atomic operations`.

Define Specific Operations as Atomic

You might want to define a group of operations as atomic. This group of operations cannot be interrupted by operations in another thread or task.

Use one of these techniques:

- **Critical sections**

Protect a group of operations with critical sections.

A critical section begins and ends with calls to specific functions. You can use a predefined set of primitives to begin or end critical sections, or use your own functions.

A group of operations in a critical section are atomic with respect to another group of operations that are in the same critical section (that is, having the same beginning and ending function).

Specify critical sections using the option `Critical section details (-critical-section-begin -critical-section-end)`.

- **Temporally exclusive tasks**

Protect a group of operations by specifying certain tasks as temporally exclusive.

If a group of tasks are temporally exclusive, all operations in one task are atomic with respect to operations in the other tasks.

Specify temporal exclusion using the option `Temporally exclusive tasks (-temporal-exclusions-file)`.

- **Task priorities** (Bug Finder only)

Protect a group of operations by specifying that certain tasks have higher priorities. For instance, interrupts have higher priorities over cyclic tasks.

You can specify up to four different priorities with these options (with highest priority listed first):

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

All operations in a task with higher priority are atomic with respect to operations in tasks with lower priorities. See also “Define Preemptable Interrupts and Nonpreemptable Tasks” (Polyspace Bug Finder).

- **Routine disabling interrupts** (Bug Finder only)

Protect a group of operations by disabling all interrupts. Use the option `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

After you call a routine to disable interrupts, all subsequent operations are atomic until you call another routine to reenable interrupts. The operations are atomic with respect to operations in all other tasks.

For a tutorial, see “Protections for Shared Variables in Multitasking Code” on page 11-20.

See Also

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

More About

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

Define Preemptable Interrupts and Nonpreemptable Tasks

Bug Finder detects data races between concurrent tasks. Using Bug Finder analysis options, you can fix data race detection by specifying that certain tasks have higher priorities over others. A task with higher priority is atomic with respect to tasks with lower priority and cannot be interrupted by those tasks.

Emulating Task Priorities

You can specify up to four different priorities with these options (with highest priority listed first):

- Interrupts (nonpreemptable): Use option `Interrupts (-interrupts)`.
- Interrupts (preemptable): Use options `Interrupts (-interrupts)` and `-preemptable-interrupts`.
- Cyclic tasks (nonpreemptable): Use options `Cyclic tasks (-cyclic-tasks)` and `-non-preemptable-tasks`.

You can also define preemptable noncyclic tasks with the option `Entry points (-entry-points)` and `-non-preemptable-tasks`.

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

You can also define noncyclic tasks with the option `Entry points (-entry-points)`.

For instance, interrupts have the highest priority and cannot be preempted by other tasks. To define a class of interrupts that can be preempted, lower their priority by making them preemptable.

Examples of Task Priorities

Consider this example with three tasks. A variable `var` is shared between the two tasks `task1` and `task2` without any protection such as a critical section. Depending on the priorities of `task1` and `task2`, Bug Finder shows a data race. The third task is not relevant for the example (and is added only to include a critical section, otherwise data race detection is disabled).

```
int var;

void begin_critical_section(void);
void end_critical_section(void);

void task1(void) {
    var++;
}

void task2(void) {
    var=0;
}

void task3(void){
    begin_critical_section();
    /* Some atomic operation */
}
```

```
    end_critical_section();  
}
```

Adjust the priorities of `task1` and `task2` and see whether a data race is detected. For instance:

1 Configure these multitasking options:

- `Interrupts (-interrupts)`: Specify `task1` and `task2` as interrupts.
- `Cyclic tasks (-cyclic-tasks)`: Specify `task3` as a cyclic task.
- `Critical section details (-critical-section-begin -critical-section-end)`: Specify `begin_critical_section` as a function beginning a critical section and `end_critical_section` as a function ending a critical section.

2 Run Bug Finder.

You do not see a data race. Since `task1` and `task2` are nonpreemptable interrupts, the shared variable cannot be accessed concurrently.

3 Change `task1` to a preemptable interrupt by using the option `-preemptable-interrupts`.

4 Run Bug Finder again. You now see a data race on the shared variable `var`.

Further Explorations

Modify this example in the following ways and see the effect of the modification:

- Change the priorities of `task1` and `task2`.

For instance, you can leave `task1` as a nonpreemptable interrupt but change `task2` to a preemptable interrupt by using the option `-preemptable-interrupts`.

The data race disappears. The reason is:

- `task1` has higher priority and cannot be interrupted by `task2`.
- The operation in `task2` is atomic and cannot be interrupted by `task1`.
- Enable the checker `Data race including atomic operations` (not enabled by default). Use the option `Find defects (-checkers)`.

You see the data race again. The checker considers all operations as potentially nonatomic and the operation in `task2` can now be interrupted by the higher priority operation in `task1`.

Try other modifications to the analysis options and see the result of the checkers.

See Also

Polyspace Analysis Options

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

Polyspace Results

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

More About

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

Define Critical Sections with Functions That Take Arguments

When verifying multitasking code, Polyspace considers that a critical section lies between calls to a lock and unlock function.

```
lock();
/* Critical section code */
unlock();
```

A group of operations in a critical section are atomic with respect to another group of operations that are in the same critical section (that is, having the same lock and unlock function). See also “Define Atomic Operations in Multitasking Code” on page 11-24.

Polyspace Assumption on Functions Defining Critical Sections

Polyspace ignores arguments to functions that begin and end critical sections.

For instance, Polyspace treats the two code sections below as the same critical section if you specify `my_task_1` and `my_task_2` as entry points, `my_lock` as the lock function and `my_unlock` as the unlock function.

```
int shared_var;

void my_lock(int);
void my_unlock(int);

void my_task_1() {
    my_lock(1);
    /* Critical section code */
    shared_var=0;
    my_unlock(1);
}

void my_task_2() {
    my_lock(2);
    /* Critical section code */
    shared_var++;
    my_unlock(2);
}
```

As a result, the analysis considers that these two sections are protected from interrupting each other even though they might not be protected. For instance, Bug Finder does not detect the data race on `shared_var`.

Often, the function arguments can be determined only at run time. Since Polyspace models the critical sections prior to the static analysis and run-time error checking phase, the analysis cannot determine if the function arguments are different and ignores the arguments.

Adapt Polyspace Analysis to Lock and Unlock Functions with Arguments

When the arguments to the functions defining critical sections are compile-time constants, you can adapt the analysis to work around the Polyspace assumption.

For instance, you can use Polyspace analysis options so that the code in the preceding example appears to Polyspace as shown here.

```
int shared_var;

void my_lock_1(void);
void my_lock_2(void);
void my_unlock_1(void);
void my_unlock_2(void);

void my_task_1() {
    my_lock_1();
    /* Critical section code */
    shared_var=0;
    my_unlock_1();
}

void my_task_2() {
    my_lock_2();
    /* Critical section code */
    shared_var++;
    my_unlock_2();
}
```

If you then specify `my_lock_1` and `my_lock_2` as the lock functions and `my_unlock_1` and `my_unlock_2` as the unlock functions, the analysis recognizes the two sections of code as part of different critical sections. For instance, Bug Finder detects a data race on `shared_var`.

To adapt the analysis for lock and unlock functions that take compile-time constants as arguments:

- 1 In a header file `common_polyspace_include.h`, convert the function arguments into extensions of the function name with `#define`-s. In addition, provide a declaration for the new functions.

For instance, for the preceding example, use these `#define`-s and declarations:

```
#define my_lock(X) my_lock_##X()
#define my_unlock(X) my_unlock_##X()

void my_lock_1(void);
void my_lock_2(void);
void my_unlock_1(void);
void my_unlock_2(void);
```

- 2 Specify the file name `common_polyspace_include.h` as argument for the option `Include (-include)`.

The analysis considers this header file as `#include`-d in all source files that are analyzed.

- 3 Specify the new function names as functions beginning and ending critical sections. Use the options `Critical section details (-critical-section-begin -critical-section-end)`.

See Also

`Critical section details (-critical-section-begin -critical-section-end)`

More About

- “Protections for Shared Variables in Multitasking Code” on page 11-20

Configure Coding Rules Checking and Code Metrics Computation

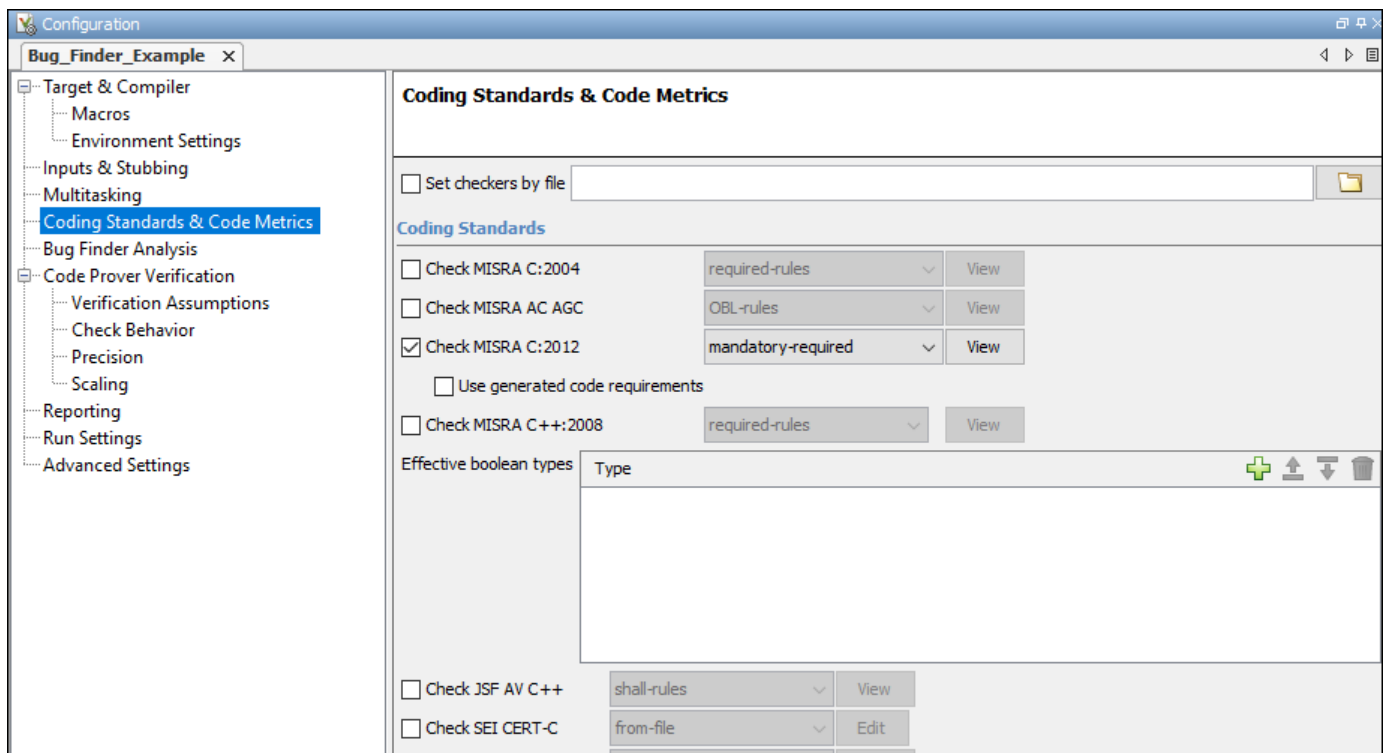
Check for Coding Standard Violations

With Polyspace, you can check your C/C++ code for violations of coding rules such as MISRA C:2012 rules. Adhering to coding rules can reduce the number of defects and improve the quality of your code.

Polyspace can detect the violations of these rules:

- MISRA C: 2004
- MISRA C: 2012
- MISRA C++
- JSF AV C++
- AUTOSAR C++14 (*Bug Finder only*)
- CERT C (*Bug Finder only*)
- CERT C++ (*Bug Finder only*)
- ISO[®]/IEC TS 17961 (*Bug Finder only*)

Configure Coding Rules Checking



Specify Standard and Predefined Checker Subsets

Specify the coding rules through Polyspace analysis options. When you run Bug Finder or Code Prover, the analysis looks for coding rule violations in addition to other checks. You can disable the other checks and look for coding rule violations only.

In the Polyspace user interface (desktop products), the options are on the **Configuration** pane under the **Coding Standards & Code Metrics** node.

For C code, use one of these options:

- Check MISRA C:2004 (-misra2)

For generated code, enable the option specific to generated code.

- Check MISRA C:2012 (-misra3)

For generated code, enable the option specific to generated code.

- Check SEI CERT-C (-cert-c)
- Check ISO/IEC TS 17961 (-iso-17961)

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

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

You can specify a predefined subset of rules, for instance, mandatory for MISRA C: 2012. These subsets are typically defined by the standard.

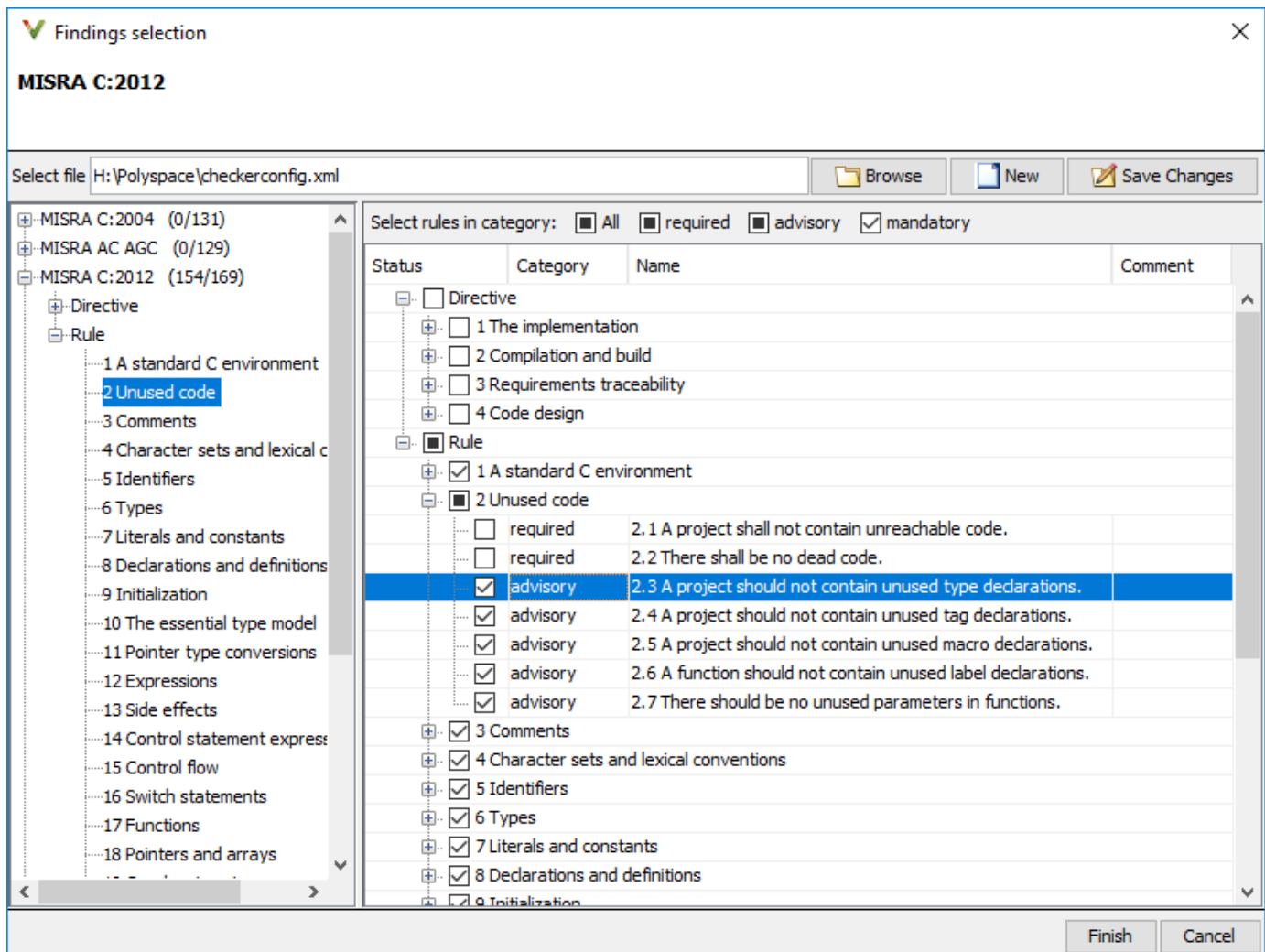
You can also define naming conventions for identifiers using regular expressions. See “Create Custom Coding Rules” on page 12-42.

Customize Checker Subsets

Instead of the predefined subsets, you can specify your own subset of rules from a coding standard.

User Interface (Desktop Products Only)

- 1 Select the coding standard. From the drop-down list for the subset of rules, select `from-file`. Click **Edit**.
- 2 In the **Findings selection** window, the coding standard is highlighted on the left pane. On the right pane, select the rules that you want to include in your analysis.



When you save the rule selections, the configuration is saved in an XML file that you can reuse for multiple analyses. The same file contains rules selected for all coding standards. You can reuse this file across multiple projects to enforce common coding standards in a team or organization. To reuse this file in another project in the Polyspace user interface:

- Choose a coding standard in the project configuration. From the drop-down list for the subset of rules, select `from-file`.
- Click **Edit** and browse to the file location. Alternatively, enter the file name as argument for the option `Set checkers by file (-checkers-selection-file)`.

Command Line

With the Polyspace desktop products, you can create a coding standard XML file in the user interface and then use this file for command-line analysis. Provide this XML file with the option `Set checkers by file (-checkers-selection-file)`.

With the Polyspace Server products, you have to create a coding standard XML from scratch. Depending on the standard that you want to enable, make a writeable copy of one of the files in *polyspaceserverroot*\help\toolbox\polyspace_bug_finder_server\examples\coding_standards_XML and turn off rules using entries in the XML file (all rules from a standard are enabled in the template). Here, *polyspaceserverroot* is the root installation folder for the Polyspace Server products, for instance, C:\Program Files\Polyspace Server\R2019a.

For instance, to turn off MISRA C: 2012 rule 8.1, use this entry in a copy of the file *misra_c_2012_rules.xml*:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
    ...
    <check id="8.1" state="off">
      </check>
    ...
  </section>
  ...
</standard>
```

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”
- “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

To check for coding standards only:

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

These rules are checked in the later stages of a Code Prover analysis: MISRA C:2004 rules 9.1, 13.7, and 21.1, and MISRA C:2012 rules 2.2, 9.1, 14.3, and 18.1. If you stop Code Prover at source

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

Review Coding Rule Violations

Result Details

Variable trace

Result Review

Status: To fix

Severity: Medium

MISRA C:2012 5.1 (Required) ?
 External identifiers shall be distinct.
 External function `demo_corrected_sighandlerasynccunsafestrict` conflicts with the external identifier `demo_corrected_sighandlerasynccunsafe` (`programming.c` line 1171).


	Event	File	Scope	Line
1	Violation site	<code>programming.c</code>	<code>programming.c</code>	1171
2	▼ MISRA C:2012 5.1	<code>programming.c</code>	File Scope	1230

Source

```

programming.c x
1226 void Corrected_sighandlerasynccunsafestrict(int signum) {
1227     int s0 = signum; /* Fix: avoid raise() */
1228 }
1229
1230 int demo_corrected_sighandlerasynccunsafestrict(void) {
1231     if (signal(SIGTERM, demo_term_handler) == SIG_ERR) {
1232         /* Handle error */
1233     }
1234     if (signal(SIGINT, corrected_sighandlerasynccunsafestrict) == SIG_ERR) {
1235         /* Handle error */
1236     }
1237     /* Program code */
1238     if (raise(SIGINT) != 0) {
1239         /* Handle error */
1240     }
1241     /* More code */
1242     return 0;
1243 }
  
```

After analysis, you see the coding standard violations on the **Results List** pane. Select a violation to see further details on the **Result Details** pane and the source code on the **Source** pane.

Violations of coding standards are indicated in the source code with the  icon.

For further steps, see “Review Analysis Results”.

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 Polyspace Code Prover Results” on page 16-2
- “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2
- “Filter and Group Results” on page 19-2
- “Generate Reports” on page 20-2

Avoid Violations of MISRA C 2012 Rules 8.x

MISRA C:2012 rules 8.1-8.14 enforce good coding practices surrounding declarations and definitions. If you follow these practices, you are less likely to have conflicting declarations or to unintentionally modify variables.

If you do not follow these practices *during coding*, your code might require major changes later to be MISRA C-compliant. You might have too many MISRA C violations. Sometimes, in fixing a violation, you might violate another rule. Instead, keep these rules in mind when coding. Use the MISRA C:2012 checker to spot any issues that you might have missed.

- **Explicitly specify all data types in declarations.**

Avoid implicit data types like this declaration of `k`:

```
extern void foo (char c, const k);
```

Instead use:

```
extern void foo (char c, const int k);
```

That way, you do not violate MISRA C:2012 Rule 8.1.

- **When declaring functions, provide names and data types for all parameters.**

Avoid declarations without parameter names like these declarations:

```
extern int func(int);  
extern int func2();
```

Instead use:

```
extern int func(int arg);  
extern int func2(void);
```

That way, you do not violate MISRA C:2012 Rule 8.2.

- **If you want to use an object or function in multiple files, declare the object or function once in only one header file.**

To use an object in multiple source files, declare it as `extern` in a header file. Include the header file in all the source files where you need the object. In one of those source files, define the object. For instance:

```
/* header.h */  
extern int var;  
  
/* file1.c */  
#include "header.h"  
/* Some usage of var */  
  
/* file2.c */  
#include "header.h"  
int var=1;
```

To use a function in multiple source files, declare it in a header file. Include the header file in all the source files where you need the function. In one of those source files, define the function.

That way, you do not violate MISRA C:2012 Rule 8.3, MISRA C:2012 Rule 8.4, MISRA C:2012 Rule 8.5, or MISRA C:2012 Rule 8.6.

- **If you want to use an object or function in one file only, declare and define the object or function with the static specifier.**

Make sure that you use the `static` specifier in all declarations and the definition. For instance, this function `func` is meant to be used only in the current file:

```
static int func(void);
static int func(void){
}
```

That way, you do not violate MISRA C:2012 Rule 8.7 and MISRA C:2012 Rule 8.8.

- **If you want to use an object in one function only, declare the object in the function body.**

Avoid declaring the object outside the function.

For instance, if you use `var` in `func` only, do declare it outside the body of `func`:

```
int var;
void func(void) {
    var=1;
}
```

Instead use:

```
void func(void) {
    int var;
    var=1;
}
```

That way, you do not violate MISRA C:2012 Rule 8.7 and MISRA C:2012 Rule 8.9.

- **If you want to inline a function, declare and define the function with the static specifier.**

Every time you add `inline` to a function definition, add `static` too:

```
static inline double func(int val);
static inline double func(int val) {
}
```

That way, you do not violate MISRA C:2012 Rule 8.10.

- **When declaring arrays, explicitly specify their size.**

Avoid implicit size specifications like this:

```
extern int32_t array[];
```

Instead use:

```
#define MAXSIZE 10
extern int32_t array[MAXSIZE];
```

That way, you do not violate MISRA C:2012 Rule 8.11.

- **When declaring enumerations, try to avoid mixing implicit and explicit specifications.**

Avoid mixing implicit and explicit specifications. You can specify the first enumeration constant explicitly, but after that, use either implicit or explicit specifications. For instance, avoid this type of mix:

```
enum color {red = 2, blue, green = 3, yellow};
```

Instead use:

```
enum color {red = 2, blue, green, yellow};
```

That way, you do not violate MISRA C:2012 Rule 8.12.

- **When declaring pointers, point to a const-qualified type unless you want to use the pointer to modify an object.**

Point to a const-qualified type by default unless you intend to use the pointer for modifying the pointed object. For instance, in this example, `ptr` is not used to modify the pointed object:

```
char last_char(const char * const ptr){  
}
```

That way, you do not violate MISRA C:2012 Rule 8.13.

Software Quality Objective Subsets (C:2004)

In this section...
“Rules in SQO-Subset1” on page 12-11
“Rules in SQO-Subset2” on page 12-12

Rules in SQO-Subset1

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

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
12.12	The underlying bit representations of floating-point values shall not be used.
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type.
13.5	The three expressions of a <i>for</i> statement shall be concerned only with loop control.
14.4	The <i>goto</i> statement shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	The declaration of objects should contain no more than 2 levels of pointer indirection.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.3	An area of memory shall not be reused for unrelated purposes.
18.4	Unions shall not be used.

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

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

Rules in SQO-Subset2

Good design practices generally lead to less code complexity, which can reduce the number of unproven results in Polyspace Code Prover. The following set of coding rules enforce good design practices. The SQO-subset2 option checks the rules in SQO-subset1 and some additional rules.

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
6.3	<i>typedefs</i> that indicate size and signedness should be used in place of the basic types
8.7	Objects shall be defined at block scope if they are only accessed from within a single function
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures
9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression
10.5	Bitwise operations shall not be performed on signed integer types
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
11.5	Type casting from any type to or from pointers shall not be used
12.1	Limited dependence should be placed on C's operator precedence rules in expressions
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits
12.5	The operands of a logical && or shall be primary-expressions

Rule number	Description
12.6	Operands of logical operators (&&, and !) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (&&, or !)
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned
12.10	The comma operator shall not be used
12.12	The underlying bit representations of floating-point values shall not be used.
13.1	Assignment operators shall not be used in expressions that yield Boolean values
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type.
13.5	The three expressions of a <i>for</i> statement shall be concerned only with loop control.
13.6	Numeric variables being used within a " <i>for</i> " loop for iteration counting should not be modified in the body of the loop
14.4	The <i>goto</i> statement shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
14.8	The statement forming the body of a <i>switch</i> , <i>while</i> , <i>do while</i> or <i>for</i> statement shall be a compound statement
14.10	All <i>if else if</i> constructs should contain a final <i>else</i> clause
15.3	The final clause of a <i>switch</i> statement shall be the <i>default</i> clause
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration
16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	The declaration of objects should contain no more than 2 levels of pointer indirection.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.

Rule number	Description
18.3	An area of memory shall not be reused for unrelated purposes.
18.4	Unions shall not be used.
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.
20.3	The validity of values passed to library functions shall be checked.
20.4	Dynamic heap memory allocation shall not be used.

Note Polyspace software does not check MISRA rule **20.3** directly.

However, you can check this rule by writing manual stubs that check the validity of values. For example, the following code checks the validity of an input being greater than 1:

```
int my_system_library_call(int in) {assert (in>1); if random \  
return -1 else return 0; }
```

See Also

Check MISRA C:2004 (-misra2)

More About

- “Check for Coding Rule Violations” on page 5-14

Software Quality Objective Subsets (AC AGC)

In this section...
“Rules in SQO-Subset1” on page 12-15
“Rules in SQO-Subset2” on page 12-15

Rules in SQO-Subset1

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

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
12.12	The underlying bit representations of floating-point values shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.4	Unions shall not be used.

For more information about these rules, see *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

Rules in SQO-Subset2

Good design practices generally lead to less code complexity, which can reduce the number of unproven results in Polyspace Code Prover. The following set of coding rules enforce good design practices. The SQO-subset2 option checks the rules in SQO-subset1 and some additional rules.

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

Rule number	Description
6.3	<i>typedefs</i> that indicate size and signedness should be used in place of the basic types
8.7	Objects shall be defined at block scope if they are only accessed from within a single function
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
11.5	Type casting from any type to or from pointers shall not be used
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned
12.10	The comma operator shall not be used
12.12	The underlying bit representations of floating-point values shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.4	Unions shall not be used.
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##

Rule number	Description
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in <code>#ifdef</code> and <code>#ifndef</code> preprocessor directives and the <code>defined()</code> operator
19.12	There shall be at most one occurrence of the <code>#</code> or <code>##</code> preprocessor operators in a single macro definition.
20.3	The validity of values passed to library functions shall be checked.

Note Polyspace software does not check MISRA rule **20.3** directly.

However, you can check this rule by writing manual stubs that check the validity of values. For example, the following code checks the validity of an input being greater than 1:

```
int my_system_library_call(int in) {assert (in>1); if random \  
return -1 else return 0; }
```

For more information about these rules, see *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

See Also

Check MISRA AC AGC (-misra-ac-agc)

More About

- “Check for Coding Rule Violations” on page 5-14

Software Quality Objective Subsets (C:2012)

In this section...
“Guidelines in SQO-Subset1” on page 12-18
“Guidelines in SQO-Subset2” on page 12-19

These subsets of MISRA C:2012 guidelines can have a direct or indirect impact on the precision of your Polyspace results. When you set up coding rules checking, you can select these subsets.

Guidelines in SQO-Subset1

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

Rule	Description
8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage
8.11	When an array with external linkage is declared, its size should be explicitly specified
8.13	A pointer should point to a const-qualified type whenever possible
11.1	Conversions shall not be performed between a pointer to a function and any other type
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type
11.4	A conversion should not be performed between a pointer to object and an integer type
11.5	A conversion should not be performed from pointer to void into pointer to object
11.6	A cast shall not be performed between pointer to void and an arithmetic type
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type
14.1	A loop counter shall not have essentially floating type
14.2	A for loop shall be well-formed
15.1	The goto statement should not be used
15.2	The goto statement shall jump to a label declared later in the same function
15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement
15.5	A function should have a single point of exit at the end
17.1	The features of <starg.h> shall not be used
17.2	Functions shall not call themselves, either directly or indirectly
18.3	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object
18.4	The +, -, += and -= operators should not be applied to an expression of pointer type

Rule	Description
18.5	Declarations should contain no more than two levels of pointer nesting
18.6	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist
19.2	The union keyword should not be used
21.3	The memory allocation and deallocation functions of <stdlib.h> shall not be used

Guidelines in SQO-Subset2

Good design practices generally lead to less code complexity, which can reduce the number of unproven results in Polyspace Code Prover. The following set of coding rules enforce good design practices. The SQO-subset2 option checks the rules in SQO-subset1 and some additional rules.

Rule	Description
8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage
8.11	When an array with external linkage is declared, its size should be explicitly specified
8.13	A pointer should point to a const-qualified type whenever possible
11.1	Conversions shall not be performed between a pointer to a function and any other type
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type
11.4	A conversion should not be performed between a pointer to object and an integer type
11.5	A conversion should not be performed from pointer to void into pointer to object
11.6	A cast shall not be performed between pointer to void and an arithmetic type
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type
11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer
12.1	The precedence of operators within expressions should be made explicit
12.3	The comma operator should not be used
13.2	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders
13.4	The result of an assignment operator should not be used
14.1	A loop counter shall not have essentially floating type
14.2	A for loop shall be well-formed
14.4	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type
15.1	The goto statement should not be used
15.2	The goto statement shall jump to a label declared later in the same function

Rule	Description
15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement
15.5	A function should have a single point of exit at the end
15.6	The body of an iteration- statement or a selection- statement shall be a compound-statement
15.7	All if ... else if constructs shall be terminated with an else statement
16.4	Every switch statement shall have a default label
16.5	A default label shall appear as either the first or the last switch label of a switch statement
17.1	The features of <starg.h> shall not be used
17.2	Functions shall not call themselves, either directly or indirectly
17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression
18.3	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object
18.4	The +, -, += and -= operators should not be applied to an expression of pointer type
18.5	Declarations should contain no more than two levels of pointer nesting
18.6	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist
19.2	The union keyword should not be used
20.4	A macro shall not be defined with the same name as a keyword
20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument
20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses
20.9	All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation
20.11	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator
21.3	The memory allocation and deallocation functions of <stdlib.h> shall not be used

See Also

Check MISRA C:2012 (-misra3)

More About

- “Check for Coding Rule Violations” on page 5-14

Software Quality Objective Subsets (C++)

In this section...

“SQO Subset 1 - Direct Impact on Selectivity” on page 12-21

“SQO Subset 2 - Indirect Impact on Selectivity” on page 12-22

SQO Subset 1 - Direct Impact on Selectivity

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

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-2	The One Definition Rule shall not be violated.
3-9-3	The underlying bit representations of floating-point values shall not be used.
5-0-15	Array indexing shall be the only form of pointer arithmetic.
5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.
6-6-5	A function shall have a single point of exit at the end of the function.
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

MISRA C++ Rule	Description
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.
7-5-4	Functions should not call themselves, either directly or indirectly.
8-4-1	Functions shall not be defined using the ellipsis notation.
9-5-1	Unions shall not be used.
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.
10-1-3	An accessible base class shall not be both virtual and nonvirtual in the same hierarchy.
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
18-4-1	Dynamic heap memory allocation shall not be used.

SQO Subset 2 - Indirect Impact on Selectivity

Good design practices generally lead to less code complexity, which can improve the number of unproven results in Polyspace Code Prover. The following set of coding rules may help to address design issues in your code. The SQO-subset2 option checks the rules in SQO-subset1 and SQO-subset2.

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

MISRA C++ Rule	Description
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.
3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.
3-9-2	typedefs that indicate size and signedness should be used in place of the basic numerical types.
3-9-3	The underlying bit representations of floating-point values shall not be used.
4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, , !, the equality operators == and !=, the unary & operator, and the conditional operator.
5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.
5-0-2	Limited dependence should be placed on C++ operator precedence rules in expressions.
5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression.
5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.
5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.
5-0-10	If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.
5-0-13	The condition of an if-statement and the condition of an iteration- statement shall have type bool
5-0-15	Array indexing shall be the only form of pointer arithmetic.
5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.
5-2-1	Each operand of a logical && or shall be a postfix - expression.
5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.
5-2-5	A cast shall not remove any const or volatile qualification from the type of a pointer or reference.
5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.
5-2-7	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.

MISRA C++ Rule	Description
5-2-11	The comma operator, && operator and the operator shall not be overloaded.
5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
5-3-3	The unary & operator shall not be overloaded.
5-18-1	The comma operator shall not be used.
6-2-1	Assignment operators shall not be used in sub-expressions.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.
6-4-2	All if ... else if constructs shall be terminated with an else clause.
6-4-6	The final clause of a switch statement shall be the default-clause.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.
6-6-5	A function shall have a single point of exit at the end of the function.
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.
7-5-4	Functions should not call themselves, either directly or indirectly.
8-4-1	Functions shall not be defined using the ellipsis notation.
8-4-3	All exit paths from a function with non- void return type shall have an explicit return statement with an expression.
8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &.
8-5-2	Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures.
8-5-3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.
9-5-1	Unions shall not be used.
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.

MISRA C++ Rule	Description
10-1-3	An accessible base class shall not be both virtual and nonvirtual in the same hierarchy.
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.
11-0-1	Member data in non- POD class types shall be private.
12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor.
12-8-2	The copy assignment operator shall be declared protected or private in an abstract class.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.
16-0-7	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.
16-2-2	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.
16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition.
18-4-1	Dynamic heap memory allocation shall not be used.

See Also

Check MISRA C++:2008 (-misra-cpp)

More About

- “Check for Coding Rule Violations” on page 5-14

Coding Rule Subsets Checked Early in Analysis

In the initial compilation phase of the analysis, Polyspace checks those coding rules that do not require the run-time error detection part of the analysis. If you want only those rules checked, you can perform a much quicker analysis.

The software provides two predefined subsets of rules that it checks earlier in the analysis. The subsets are available with the options Check MISRA C:2004 (-misra2), Check MISRA AC AGC (-misra-ac-agc), and Check MISRA C:2012 (-misra3).

Argument	Purpose
single-unit-rules	<p>Check rules that apply only to single translation units.</p> <p>If you detect only coding rule violations and select this subset, a Bug Finder analysis stops after the compilation phase.</p>
system-decidable-rules	<p>Check rules in the single-unit-rules subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit.</p> <p>If you detect only coding rule violations and select this subset, a Bug Finder analysis stops after the linking phase.</p>

See also “Check for Coding Rule Violations” on page 5-14.

MISRA C: 2004 and MISRA AC AGC Rules

The software checks the following rules early in the analysis. The rules that are checked at a system level and appear only in the system-decidable-rules subset are indicated by an asterisk.

Environment

Rule	Description
1.1*	All code shall conform to ISO 9899:1990 "Programming languages - C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.

Language Extensions

Rule	Description
2.1	Assembly language shall be encapsulated and isolated.
2.2	Source code shall only use /* */ style comments.
2.3	The character sequence /* shall not be used within a comment.

Documentation

Rule	Description
3.4	All uses of the #pragma directive shall be documented and explained.

Character Sets

Rule	Description
4.1	Only those escape sequences which are defined in the ISO C standard shall be used.
4.2	Trigraphs shall not be used.

Identifiers

Rule	Description
5.1*	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
5.3*	A typedef name shall be a unique identifier.
5.4*	A tag name shall be a unique identifier.
5.5*	No object or function identifier with a static storage duration should be reused.
5.6*	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.
5.7*	No identifier name should be reused.

Types

Rule	Description
6.1	The plain char type shall be used only for the storage and use of character values.
6.2	Signed and unsigned char type shall be used only for the storage and use of numeric values.
6.3	typedefs that indicate size and signedness should be used in place of the basic types.
6.4	Bit fields shall only be defined to be of type <code>unsigned int</code> or <code>signed int</code> .
6.5	Bit fields of type <code>signed int</code> shall be at least 2 bits long.

Constants

Rule	Description
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and Definitions

Rule	Description
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated.
8.3	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.
8.4*	If objects or functions are declared more than once their types shall be compatible.
8.5	There shall be no definitions of objects or functions in a header file.
8.6	Functions shall always be declared at file scope.
8.7	Objects shall be defined at block scope if they are only accessed from within a single function.
8.8*	An external object or function shall be declared in one file and only one file.
8.9*	An identifier with external linkage shall have exactly one external definition.
8.10*	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.
8.11	The <code>static</code> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

Initialization

Rule	Description
9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.
9.3	In an enumerator list, the <code>=</code> construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Arithmetic Type Conversion

Rule	Description
10.1	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: <ul style="list-style-type: none"> • It is not a conversion to a wider integer type of the same signedness, or • The expression is complex, or • The expression is not constant and is a function argument, or • The expression is not constant and is a return expression
10.2	The value of an expression of floating type shall not be implicitly converted to a different type if <ul style="list-style-type: none"> • It is not a conversion to a wider floating type, or • The expression is complex, or • The expression is a function argument, or • The expression is a return expression
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.
10.4	The value of a complex expression of float type may only be cast to narrower floating type.
10.5	If the bitwise operator <code>~</code> and <code><<</code> are applied to an operand of underlying type <code>unsigned char</code> or <code>unsigned short</code> , the result shall be immediately cast to the underlying type of the operand
10.6	The "U" suffix shall be applied to all constants of <code>unsigned</code> types.

Pointer Type Conversion

Rule	Description
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to <code>void</code> .
11.3	A cast should not be performed between a pointer type and an integral type.
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.
11.5	A cast shall not be performed that removes any <code>const</code> or <code>volatile</code> qualification from the type addressed by a pointer

Expressions

Rule	Description
12.1	Limited dependence should be placed on C's operator precedence rules in expressions.
12.3	The <code>sizeof</code> operator should not be used on expressions that contain side effects.
12.5	The operands of a logical <code>&&</code> or <code> </code> shall be primary-expressions.
12.6	Operands of logical operators (<code>&&</code> , <code> </code> and <code>!</code>) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (<code>&&</code> , <code> </code> or <code>!</code>).
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed.
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
12.10	The comma operator shall not be used.
12.11	Evaluation of constant unsigned expression should not lead to wraparound.
12.12	The underlying bit representations of floating-point values shall not be used.
12.13	The increment (<code>++</code>) and decrement (<code>--</code>) operators should not be mixed with other operators in an expression

Control Statement Expressions

Rule	Description
13.1	Assignment operators shall not be used in expressions that yield Boolean values.
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a <code>for</code> statement shall not contain any objects of floating type.
13.5	The three expressions of a <code>for</code> statement shall be concerned only with loop control.
13.6	Numeric variables being used within a <code>for</code> loop for iteration counting should not be modified in the body of the loop.

Control Flow

Rule	Description
14.3	All non-null statements shall either <ul style="list-style-type: none"> • have at least one side effect however executed, or • cause control flow to change.
14.4	The <code>goto</code> statement shall not be used.
14.5	The <code>continue</code> statement shall not be used.
14.6	For any iteration statement, there shall be at most one <code>break</code> statement used for loop termination.
14.7	A function shall have a single point of exit at the end of the function.
14.8	The statement forming the body of a <code>switch</code> , <code>while</code> , <code>do while</code> or <code>for</code> statement shall be a compound statement.
14.9	An <code>if (expression)</code> construct shall be followed by a compound statement. The <code>else</code> keyword shall be followed by either a compound statement, or another <code>if</code> statement.
14.10	All <code>if else if</code> constructs should contain a final <code>else</code> clause.

Switch Statements

Rule	Description
15.0	Unreachable code is detected between <code>switch</code> statement and first <code>case</code> .
15.1	A <code>switch</code> label shall only be used when the most closely-enclosing compound statement is the body of a <code>switch</code> statement
15.2	An unconditional <code>break</code> statement shall terminate every non-empty <code>switch</code> clause.
15.3	The final clause of a <code>switch</code> statement shall be the <code>default</code> clause.
15.4	A <code>switch</code> expression should not represent a value that is effectively Boolean.
15.5	Every <code>switch</code> statement shall have at least one <code>case</code> clause.

Functions

Rule	Description
16.1	Functions shall not be defined with variable numbers of arguments.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.
16.4*	The identifiers used in the declaration and definition of a function shall be identical.
16.5	Functions with no parameters shall be declared with parameter type <code>void</code> .
16.6	The number of arguments passed to a function shall match the number of parameters.
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
16.9	A function identifier shall only be used with either a preceding <code>&</code> , or with a parenthesized parameter list, which may be empty.

Pointers and Arrays

Rule	Description
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	A type should not contain more than 2 levels of pointer indirection.

Structures and Unions

Rule	Description
18.1	All structure or union types shall be complete at the end of a translation unit.
18.4	Unions shall not be used.

Preprocessing Directives

Rule	Description
19.1	<code>#include</code> statements in a file shall only be preceded by other preprocessors directives or comments.
19.2	Nonstandard characters should not occur in header file names in <code>#include</code> directives.
19.3	The <code>#include</code> directive shall be followed by either a <code><filename></code> or "filename" sequence.
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.
19.5	Macros shall not be <code>#defined</code> and <code>#undefd</code> within a block.
19.6	<code>#undef</code> shall not be used.
19.7	A function should be used in preference to a function like-macro.
19.8	A function-like macro shall not be invoked without all of its arguments.
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
19.10	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of <code>#</code> or <code>##</code> .
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in <code>#ifdef</code> and <code>#ifndef</code> preprocessor directives and the <code>defined()</code> operator.
19.12	There shall be at most one occurrence of the <code>#</code> or <code>##</code> preprocessor operators in a single macro definition.
19.13	The <code>#</code> and <code>##</code> preprocessor operators should not be used.
19.14	The <code>defined</code> preprocessor operator shall only be used in one of the two standard forms.
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.
19.17	All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> or <code>#ifdef</code> directive to which they are related.

Standard Libraries

Rule	Description
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.
20.2	The names of standard library macros, objects and functions shall not be reused.
20.4	Dynamic heap memory allocation shall not be used.
20.5	The error indicator <code>errno</code> shall not be used.
20.6	The macro <code>offsetof</code> , in library <code><stddef.h></code> , shall not be used.
20.7	The <code>setjmp</code> macro and the <code>longjmp</code> function shall not be used.
20.8	The signal handling facilities of <code><signal.h></code> shall not be used.
20.9	The input/output library <code><stdio.h></code> shall not be used in production code.
20.10	The library functions <code>atof</code> , <code>atoi</code> and <code>atoll</code> from library <code><stdlib.h></code> shall not be used.
20.11	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> from library <code><stdlib.h></code> shall not be used.
20.12	The time handling functions of library <code><time.h></code> shall not be used.

The rules that are checked at a system level and appear only in the `system-decidable-rules` subset are indicated by an asterisk.

MISRA C: 2012 Rules

The software checks the following rules early in the analysis. The rules that are checked at a system level and appear only in the `system-decidable-rules` subset are indicated by an asterisk.

Standard C Environment

Rule	Description
1.1	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.
1.2	Language extensions should not be used.

Unused Code

Rule	Description
2.3*	A project should not contain unused type declarations.
2.4*	A project should not contain unused tag declarations.
2.5*	A project should not contain unused macro declarations.
2.6	A function should not contain unused label declarations.
2.7	There should be no unused parameters in functions.

Comments

Rule	Description
3.1	The character sequences <code>/*</code> and <code>//</code> shall not be used within a comment.
3.2	Line-splicing shall not be used in <code>//</code> comments.

Character Sets and Lexical Conventions

Rule	Description
4.1	Octal and hexadecimal escape sequences shall be terminated.
4.2	Trigraphs should not be used.

Identifiers

Rule	Description
5.1*	External identifiers shall be distinct.
5.2	Identifiers declared in the same scope and name space shall be distinct.
5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
5.4	Macro identifiers shall be distinct.
5.5	Identifiers shall be distinct from macro names.
5.6*	A typedef name shall be a unique identifier.
5.7*	A tag name shall be a unique identifier.
5.8*	Identifiers that define objects or functions with external linkage shall be unique.
5.9*	Identifiers that define objects or functions with internal linkage should be unique.

Types

Rule	Description
6.1	Bit-fields shall only be declared with an appropriate type.
6.2	Single-bit named bit fields shall not be of a signed type.

Literals and Constants

Rule	Description
7.1	Octal constants shall not be used.
7.2	A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type.
7.3	The lowercase character "l" shall not be used in a literal suffix.
7.4	A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

Declarations and Definitions

Rule	Description
8.1	Types shall be explicitly specified.
8.2	Function types shall be in prototype form with named parameters.
8.3*	All declarations of an object or function shall use the same names and type qualifiers.
8.4	A compatible declaration shall be visible when an object or function with external linkage is defined.
8.5*	An external object or function shall be declared once in one and only one file.
8.6*	An identifier with external linkage shall have exactly one external definition.
8.7*	Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.
8.8	The <code>static</code> storage class specifier shall be used in all declarations of objects and functions that have internal linkage.
8.9*	An object should be defined at block scope if its identifier only appears in a single function.
8.10	An inline function shall be declared with the <code>static</code> storage class.
8.11	When an array with external linkage is declared, its size should be explicitly specified.
8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.
8.14	The <code>restrict</code> type qualifier shall not be used.

Initialization

Rule	Description
9.2	The initializer for an aggregate or union shall be enclosed in braces.
9.3	Arrays shall not be partially initialized.
9.4	An element of an object shall not be initialized more than once.
9.5	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

The Essential Type Model

Rule	Description
10.1	Operands shall not be of an inappropriate essential type.
10.2	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.
10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.
10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.
10.5	The value of an expression should not be cast to an inappropriate essential type.
10.6	The value of a composite expression shall not be assigned to an object with wider essential type.
10.7	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.
10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

Pointer Type Conversion

Rule	Description
11.1	Conversions shall not be performed between a pointer to a function and any other type.
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type.
11.3	A cast shall not be performed between a pointer to object type and a pointer to a different object type.
11.4	A conversion should not be performed between a pointer to object and an integer type.
11.5	A conversion should not be performed from pointer to void into pointer to object.
11.6	A cast shall not be performed between pointer to void and an arithmetic type.
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type.
11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.
11.9	The macro NULL shall be the only permitted form of integer null pointer constant.

Expressions

Rule	Description
12.1	The precedence of operators within expressions should be made explicit.
12.3	The comma operator should not be used.
12.4	Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Side Effects

Rule	Description
13.3	A full expression containing an increment (++) or decrement (- -) operator should have no other potential side effects other than that caused by the increment or decrement operator.
13.4	The result of an assignment operator should not be used.
13.6	The operand of the <code>sizeof</code> operator shall not contain any expression which has potential side effects.

Control Statement Expressions

Rule	Description
14.4	The controlling expression of an <code>if</code> statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

Control Flow

Rule	Description
15.1	The <code>goto</code> statement should not be used.
15.2	The <code>goto</code> statement shall jump to a label declared later in the same function.
15.3	Any label referenced by a <code>goto</code> statement shall be declared in the same block, or in any block enclosing the <code>goto</code> statement.
15.4	There should be no more than one <code>break</code> or <code>goto</code> statement used to terminate any iteration statement.
15.5	A function should have a single point of exit at the end
15.6	The body of an iteration-statement or a selection-statement shall be a compound statement.
15.7	All <code>if ... else if</code> constructs shall be terminated with an <code>else</code> statement.

Switch Statements

Rule	Description
16.1	All <code>switch</code> statements shall be well-formed.
16.2	A <code>switch</code> label shall only be used when the most closely-enclosing compound statement is the body of a <code>switch</code> statement.
16.3	An unconditional <code>break</code> statement shall terminate every <code>switch</code> -clause.
16.4	Every <code>switch</code> statement shall have a <code>default</code> label.
16.5	A <code>default</code> label shall appear as either the first or the last <code>switch</code> label of a <code>switch</code> statement.
16.6	Every <code>switch</code> statement shall have at least two <code>switch</code> -clauses.
16.7	A <code>switch</code> -expression shall not have essentially Boolean type.

Functions

Rule	Description
17.1	The features of <stdarg.h> shall not be used.
17.3	A function shall not be declared implicitly.
17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
17.6	The declaration of an array parameter shall not contain the <code>static</code> keyword between the [].
17.7	The value returned by a function having non-void return type shall be used.

Pointers and Arrays

Rule	Description
18.4	The +, -, += and -= operators should not be applied to an expression of pointer type.
18.5	Declarations should contain no more than two levels of pointer nesting.
18.7	Flexible array members shall not be declared.
18.8	Variable-length array types shall not be used.

Overlapping Storage

Rule	Description
19.2	The <code>union</code> keyword should not be used.

Preprocessing Directives

Rule	Description
20.1	<code>#include</code> directives should only be preceded by preprocessor directives or comments.
20.2	The <code>'</code> , <code>"</code> , or <code>\</code> characters and the <code>/*</code> or <code>//</code> character sequences shall not occur in a header file name.
20.3	The <code>#include</code> directive shall be followed by either a <code><filename></code> or <code>"filename"</code> sequence.
20.4	A macro shall not be defined with the same name as a keyword.
20.5	<code>#undef</code> should not be used.
20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument.
20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.
20.8	The controlling expression of a <code>#if</code> or <code>#elif</code> preprocessing directive shall evaluate to 0 or 1.
20.9	All identifiers used in the controlling expression of <code>#if</code> or <code>#elif</code> preprocessing directives shall be <code>#define</code> 'd before evaluation.
20.10	The <code>#</code> and <code>##</code> preprocessor operators should not be used.
20.11	A macro parameter immediately following a <code>#</code> operator shall not immediately be followed by a <code>##</code> operator.
20.12	A macro parameter used as an operand to the <code>#</code> or <code>##</code> operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.
20.13	A line whose first token is <code>#</code> shall be a valid preprocessing directive.
20.14	All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> , <code>#ifdef</code> or <code>#ifndef</code> directive to which they are related.

Standard Libraries

Rule	Description
21.1	<code>#define</code> and <code>#undef</code> shall not be used on a reserved identifier or reserved macro name.
21.2	A reserved identifier or macro name shall not be declared.
21.3	The memory allocation and deallocation functions of <code><stdlib.h></code> shall not be used.
21.4	The standard header file <code><setjmp.h></code> shall not be used.
21.5	The standard header file <code><signal.h></code> shall not be used.
21.6	The Standard Library input/output functions shall not be used.
21.7	The <code>atof</code> , <code>atoi</code> , <code>atol</code> , and <code>atoll</code> functions of <code><stdlib.h></code> shall not be used.
21.8	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> of <code><stdlib.h></code> shall not be used.
21.9	The library functions <code>bsearch</code> and <code>qsort</code> of <code><stdlib.h></code> shall not be used.
21.10	The Standard Library time and date functions shall not be used.
21.11	The standard header file <code><tgmath.h></code> shall not be used.
21.12	The exception handling features of <code><fenv.h></code> should not be used.

The rules that are checked at a system level and appear only in the `system-decidable-rules` subset are indicated by an asterisk.

See Also

Check MISRA C:2004 (`-misra2`) | Check MISRA C:2012 (`-misra3`) | Check MISRA AC AGC (`-misra-ac-agc`)

More About

- “Check for Coding Rule Violations” on page 5-14

Create Custom Coding Rules

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


The tutorial uses the following code stored in a file `printInitialValue.c`:

```
#include <stdio.h>

typedef struct {
    int a;
    int b;
} collection;

void main()
{
    collection myCollection= {0,0};
    printf("Initial values in the collection are %d and %d.",
        myCollection.a,myCollection.b);
}
```

User Interface (Desktop Products Only)

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

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

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

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

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

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

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

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

Command Line

With the Polyspace desktop products, you can create a coding standard XML file in the user interface and then use this file for command-line analysis. Provide this XML file with the option `Set checkers by file (-checkers-selection-file)`.

With the Polyspace Server products, you have to create a coding standard XML from scratch. Make a writable copy of the file `custom_rules.xml` in `polyspaceserverroot\help\toolbox\polyspace_bug_finder_server\examples\coding_standards_XML` and turn off rules using entries in the XML file (all rules from a standard are enabled in the template). Here, `polyspaceserverroot` is the root installation folder for the Polyspace Server products, for instance, `C:\Program Files\Polyspace Server\R2019a`.

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

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

Provide this file as argument for the option `Set checkers by file (-checkers-selection-file)` during analysis, along with the option `Check custom rules (-custom-rules)`. For instance, for custom rules checking with Polyspace Code Prover Server, enter:

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

See Also

Check custom rules (`-custom-rules`)

Compute Code Complexity Metrics

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

Polyspace does not compute code complexity metrics by default. To compute them during analysis, use the option `Calculate code metrics (-code-metrics)`.

After analysis, the software displays project, file and function metrics on the **Results List** pane. You can compare the computed metric values against predefined limits. If a metric value exceeds limits, you can redesign your code to lower the metric value. For instance, if the number of called functions is high and several of those functions are always called together, you can write one function that fuses the bodies of those functions. Call that one function instead of the group of functions that are called together.

Impose Limits on Metrics (Desktop Products Only)

In the user interface of the Polyspace desktop products, open some results with metrics computations. Then impose limits on the metric values and update results on the **Results List** pane to show only metric values that exceed the limits.

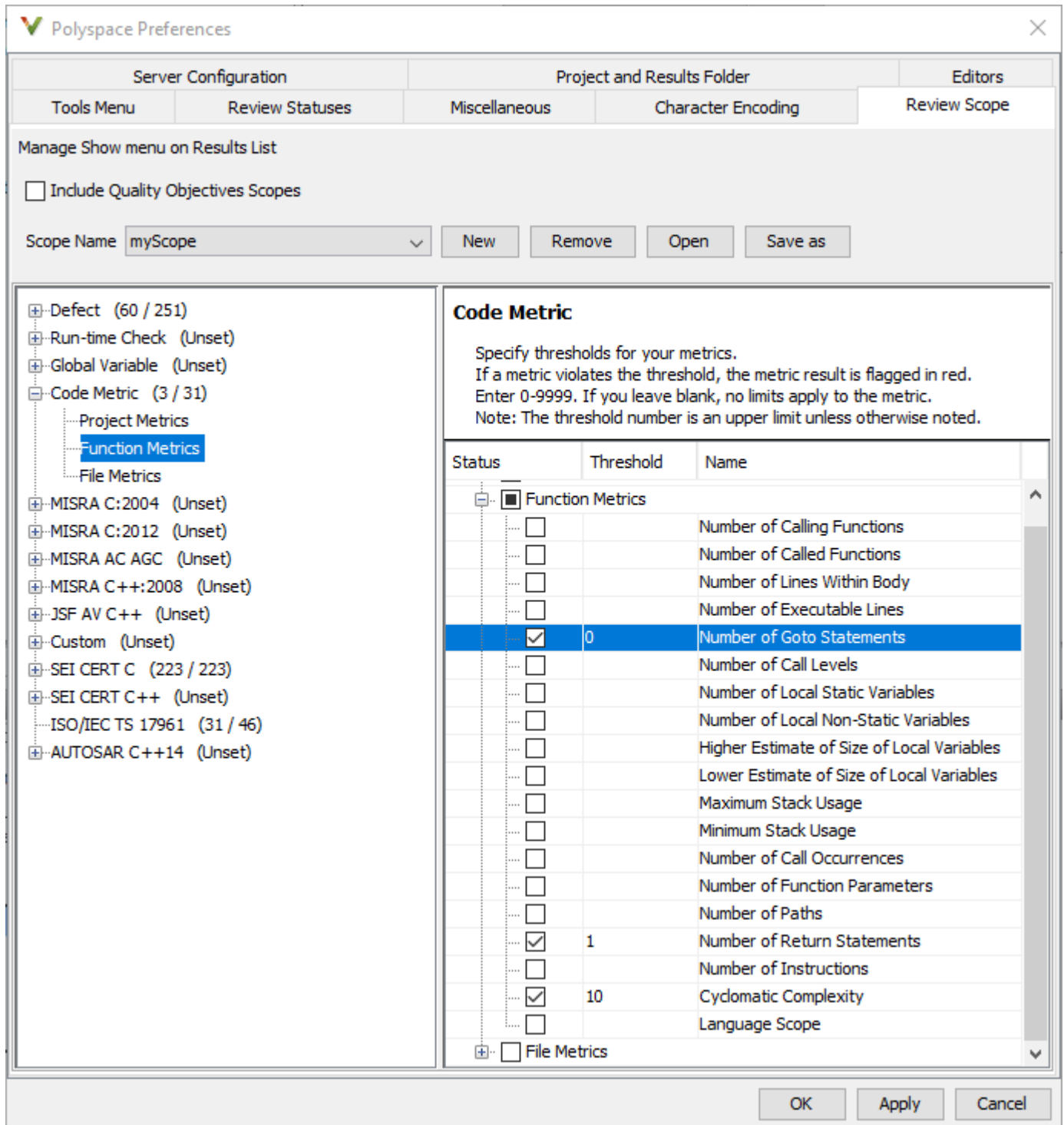
- 1 Select **Tools > Preferences**.
- 2 On the **Review Scope** tab, do one of the following:
 - To use a predefined limit, select **Include Quality Objectives Scopes**.

The **Scope Name** list shows the additional option **HIS**. The option **HIS** displays the **HIS** code metrics on page 12-47 only. Select the option to see the limit values.

- To define your own limits, select **New**. Save your limits file.

On the left pane, select **Code Metric**. On the right, select a metric and specify a limit value for the metric. Other than **Comment Density**, limit values are upper limits.

To select all metrics in a category such as **Function Metrics**, select the box next to the category name. For more information on the metrics categories, see “Code Metrics”. If only a some metrics in a category are selected, the check box next to the category name displays a symbol.



3 Select **Apply** or **OK**.

The drop-down list in the left of the **Results List** pane toolbar displays additional options.

- If you use predefined limits, the option HIS appears. This option displays code metrics only.

- If you define your own limits, the option corresponding to your limits file name appears.
- 4 Select the option corresponding to the limits that you want. Only metric values that violate your limits appear on the **Results List** pane.
 - 5 Review each violation and decide how to rework your code to avoid the violation.

Note To enforce coding standards across your organization, share your limits file that you saved in XML format.

People in your organization can use the **Open** button on the **Review Scope** tab and navigate to the location of the XML file.

Impose Limits on Metrics (Server and Access products)

In the Polyspace Access web interface, limits on code complexity metrics are predefined. In the **Dashboard** perspective, if you select **Code Metric**, a **Code Metrics** window shows the metric values and limits.

To find the limits used, see “HIS Code Complexity Metrics” on page 12-47.

See Also

Calculate code metrics (-code-metrics)

More About

- “Code Metrics”
- “HIS Code Complexity Metrics” on page 12-47

HIS Code Complexity Metrics

The following list shows the Hersteller Initiative Software (HIS) standard metrics that Polyspace evaluates. These metrics and the recommended limits for their values are part of a standard defined by a major group of Original Equipment Manufacturers or OEMs. For more information on how to focus your review to this subset of code metrics, see “Compute Code Complexity Metrics” on page 12-44.

Project

Polyspace evaluates the following HIS metrics at the project level.

Metric	Recommended Upper Limit
Number of direct recursions	0
Number of recursions	0

File

Polyspace evaluates the HIS metric, comment density, at the file level. The recommended lower limit is 20.

Function

Polyspace evaluates the following HIS metrics at the function level.

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

See Also

More About

- “Compute Code Complexity Metrics” on page 12-44
- “Code Metrics”

Coding Rule Sets and Concepts

- “Polyspace MISRA C 2004 and MISRA AC AGC Checkers” on page 13-2
- “MISRA C:2004 and MISRA AC AGC Coding Rules” on page 13-3
- “Polyspace MISRA C:2012 Checkers” on page 13-34
- “Essential Types in MISRA C: 2012 Rules 10.x” on page 13-35
- “Unsupported MISRA C:2012 Guidelines” on page 13-37
- “Polyspace MISRA C++ Checkers” on page 13-38
- “Unsupported MISRA C++ Coding Rules” on page 13-39
- “Polyspace JSF C++ Checkers” on page 13-43
- “JSF C++ Coding Rules” on page 13-44

Polyspace MISRA C 2004 and MISRA AC AGC Checkers

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

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

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

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

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

- “Software Quality Objective Subsets (C:2004)” on page 12-11
- “Software Quality Objective Subsets (AC AGC)” on page 12-15

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

See Also

More About

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

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

MISRA C:2004 and MISRA AC AGC Coding Rules

In this section...

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

“Troubleshooting” on page 13-3

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

“Unsupported MISRA C:2004 and MISRA AC AGC Rules” on page 13-32

Supported MISRA C:2004 and MISRA AC AGC Rules

The following tables list MISRA C:2004 coding rules that the Polyspace coding rules checker supports. Details regarding how the software checks individual rules and any limitations on the scope of checking are described in the “Polyspace Specification” column.

Note The Polyspace coding rules checker:

- Supports MISRA-C:2004 Technical Corrigendum 1 for rules 4.1, 5.1, 5.3, 6.1, 6.3, 7.1, 9.2, 10.5, 12.6, 13.5, and 15.0.
- Checks rules specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

The software reports most violations during the compile phase of an analysis. However, the software detects violations of rules 9.1 (Non-initialized variable), 12.11 (one of the overflow checks) using `-scalar-overflows-checks signed-and-unsigned`, 13.7 (dead code), 14.1 (dead code), 16.2 and 21.1 during code analysis, and reports these violations as run-time errors.

Note Some violations of rules 13.7 and 14.1 are reported during the compile phase of analysis.

Troubleshooting

If you expect a rule violation but do not see it, check out “Coding Standard Violations Not Displayed” on page 22-81.

List of Supported Coding Rules

- “Environment” on page 13-5
- “Language Extensions” on page 13-6
- “Documentation” on page 13-7
- “Character Sets” on page 13-7
- “Identifiers” on page 13-8
- “Types” on page 13-9
- “Constants” on page 13-9
- “Declarations and Definitions” on page 13-10

- “Initialization” on page 13-12
- “Arithmetic Type Conversion” on page 13-13
- “Pointer Type Conversion” on page 13-16
- “Expressions” on page 13-17
- “Control Statement Expressions” on page 13-19
- “Control Flow” on page 13-22
- “Switch Statements” on page 13-23
- “Functions” on page 13-24
- “Pointers and Arrays” on page 13-25
- “Structures and Unions” on page 13-26
- “Preprocessing Directives” on page 13-26
- “Standard Libraries” on page 13-29
- “Runtime Failures” on page 13-32

Environment

N.	MISRA Definition	Messages in report file	Polyspace Implementation
1.1	All code shall conform to ISO 9899:1990 "Programming languages - C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.	<p>The text <i>All code shall conform to ISO 9899:1990 Programming languages C, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996</i> precedes each of the following messages:</p> <ul style="list-style-type: none"> • ANSI C does not allow '#include_next' • ANSI C does not allow macros with variable arguments list • ANSI C does not allow '#assert' • ANSI C does not allow '#unassert' • ANSI C does not allow testing assertions • ANSI C does not allow '#ident' • ANSI C does not allow '#sccs' • text following '#else' violates ANSI standard. • text following '#endif' violates ANSI standard. • text following '#else' or '#endif' violates ANSI standard. 	All the supported extensions lead to a violation of this MISRA rule. Standard compilation error messages do not lead to a violation of this MISRA rule and remain unchanged.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
1.1 (cont.)		<p>The text <i>All code shall conform to ISO 9899:1990 Programming languages C, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996</i> precedes each of the following messages:</p> <ul style="list-style-type: none"> • ANSI C90 forbids 'long long int' type. • ANSI C90 forbids 'long double' type. • ANSI C90 forbids long long integer constants. • Keyword 'inline' should not be used. • Array of zero size should not be used. • Integer constant does not fit within unsigned long int. • Integer constant does not fit within long int. • Too many nesting levels of #includes: N_1. The limit is N_0. • Too many macro definitions: N_1. The limit is N_0. • Too many nesting levels for control flow: N_1. The limit is N_0. • Too many enumeration constants: N_1. The limit is N_0. 	

Language Extensions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
2.1	Assembly language shall be encapsulated and isolated.	Assembly language shall be encapsulated and isolated.	<p>No warnings if code is encapsulated in the following:</p> <ul style="list-style-type: none"> • asm functions or asm pragma • Macros

N.	MISRA Definition	Messages in report file	Polyspace Implementation
2.2	Source code shall only use <code>/** */</code> style comments	C++ comments shall not be used.	C++ comments are handled as comments but lead to a violation of this MISRA rule Note: This rule cannot be annotated in the source code.
2.3	The character sequence <code>/*</code> shall not be used within a comment	The character sequence <code>/*</code> shall not appear within a comment.	This rule violation is also raised when the character sequence <code>/*</code> inside a C++ comment. Note: This rule cannot be annotated in the source code.

Documentation

Rule	MISRA Definition	Messages in report file	Polyspace Implementation
3.4	All uses of the <code>#pragma</code> directive shall be documented and explained.	All uses of the <code>#pragma</code> directive shall be documented and explained.	To check this rule, you must list the pragmas that are allowed in source files by using the option <code>Allowed pragmas (-allowed-pragmas)</code> . If Polyspace finds a pragma not in the allowed pragma list, a violation is raised.

Character Sets

N.	MISRA Definition	Messages in report file	Polyspace Implementation
4.1	Only those escape sequences which are defined in the ISO C standard shall be used.	<code>\<character></code> is not an ISO C escape sequence Only those escape sequences which are defined in the ISO C standard shall be used.	
4.2	Trigraphs shall not be used.	Trigraphs shall not be used.	Trigraphs are handled and converted to the equivalent character but lead to a violation of the MISRA rule

Identifiers

N.	MISRA Definition	Messages in report file	Polyspace Implementation
5.1	Identifiers (internal and external) shall not rely on the significance of more than 31 characters	Identifier 'XX' should not rely on the significance of more than 31 characters.	All identifiers (global, static and local) are checked. For easier review, the rule checker shows all identifiers that have the same first 31 characters as one rule violation. You can see all instances of conflicting identifier names in the event history of that rule violation.
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	<ul style="list-style-type: none"> • Local declaration of XX is hiding another identifier. • Declaration of parameter XX is hiding another identifier. 	Assumes that rule 8.1 is not violated.
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
5.5	No object or function identifier with a static storage duration should be reused.	{static identifier/parameter name}'%s' should not be reused. (already used as {static identifier/parameter name} with static storage duration at %s:%d)	Warning when a static name is reused as another identifier name Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
5.6	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.	{member name}'%s' should not be reused. (already used as {member name} at %s:%d)	Warning when an idf in a namespace is reused in another namespace
5.7	No identifier name should be reused.	{identifier}'%s' should not be reused. (already used as {identifier} at %s:%d)	No violation reported when: <ul style="list-style-type: none"> • Different functions have parameters with the same name • Different functions have local variables with the same name • A function has a local variable that has the same name as a parameter of another function

Types

N.	MISRA Definition	Messages in report file	Polyspace Implementation
6.1	The plain char type shall be used only for the storage and use of character values	Only permissible operators on plain chars are '=', '==' or '!=' operators, explicit casts to integral types and '?' (for the 2nd and 3rd operands)	Warning when a plain char is used with an operator other than =, ==, !=, explicit casts to integral types, or as the second or third operands of the ? operator.
6.2	Signed and unsigned char type shall be used only for the storage and use of numeric values.	<ul style="list-style-type: none"> Value of type plain char is implicitly converted to signed char. Value of type plain char is implicitly converted to unsigned char. Value of type signed char is implicitly converted to plain char. Value of type unsigned char is implicitly converted to plain char. 	Warning if value of type plain char is implicitly converted to value of type signed char or unsigned char.
6.3	<i>typedefs</i> that indicate size and signedness should be used in place of the basic types	<i>typedefs</i> that indicate size and signedness should be used in place of the basic types.	No warning is given in typedef definition.
6.4	Bit fields shall only be defined to be of type <i>unsigned int</i> or <i>signed int</i> .	Bit fields shall only be defined to be of type unsigned int or signed int.	
6.5	Bit fields of type <i>signed int</i> shall be at least 2 bits long.	Bit fields of type signed int shall be at least 2 bits long.	No warning on anonymous signed int bitfields of width 0 - Extended to all signed bitfields of size <= 1 (if Rule 6.4 is violated).

Constants

N.	MISRA Definition	Messages in report file	Polyspace Implementation
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.	<ul style="list-style-type: none"> Octal constants other than zero and octal escape sequences shall not be used. Octal constants (other than zero) should not be used. Octal escape sequences should not be used. 	

Declarations and Definitions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	<ul style="list-style-type: none"> • Function XX has no complete prototype visible at call. • Function XX has no prototype visible at definition. 	Prototype visible at call must be complete.
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated	Whenever an object or function is declared or defined, its type shall be explicitly stated.	
8.3	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.	Definition of function 'XX' incompatible with its declaration.	Assumes that rule 8.1 is not violated. The rule is restricted to compatible types. Can be turned to Off
8.4	If objects or functions are declared more than once their types shall be compatible.	<ul style="list-style-type: none"> • If objects or functions are declared more than once their types shall be compatible. • Global declaration of 'XX' function has incompatible type with its definition. • Global declaration of 'XX' variable has incompatible type with its definition. 	Violations of this rule might be generated during the link phase. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8.5	There shall be no definitions of objects or functions in a header file	<ul style="list-style-type: none"> • Object 'XX' should not be defined in a header file. • Function 'XX' should not be defined in a header file. • Fragment of function should not be defined in a header file. 	Tentative definitions are considered as definitions. For objects with file scope, tentative definitions are declarations that: <ul style="list-style-type: none"> • Do not have initializers. • Do not have storage class specifiers, or have the <code>static</code> specifier
8.6	Functions shall always be declared at file scope.	Function 'XX' should be declared at file scope.	This rule maps to ISO/IEC TS 17961 ID addresscape.
8.7	Objects shall be defined at block scope if they are only accessed from within a single function	Object 'XX' should be declared at block scope.	Restricted to static objects.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
8.8	An external object or function shall be declared in one file and only one file	Function/Object 'XX' has external declarations in multiple files.	<p>Restricted to explicit extern declarations (tentative definitions are ignored).</p> <p>Polyspace considers that variables or functions declared <code>extern</code> in a non-header file violate this rule.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>
8.9	An identifier with external linkage shall have exactly one external definition.	<ul style="list-style-type: none"> • Procedure/Global variable XX multiply defined. • Forbidden multiple tentative definitions for object XX • Global variable has multiple tentative definitions • Undefined global variable XX 	<p>The checker flags multiple definitions only if the definitions occur in different files.</p> <p>No warnings appear on predefined symbols.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>
8.10	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required	Function/Variable XX should have internal linkage.	<p>Assumes that 8.1 is not violated. No warning if 0 uses.</p> <p>If your code does not contain a <code>main</code> function and you use options such as <code>Variables to initialize (-main-generator-writes-variables)</code> with value <code>custom</code> to explicitly specify a set of variables to initialize, the checker does not flag those variables. The checker assumes that in a real application, the file containing the <code>main</code> must initialize the variables in addition to any file that currently uses them. Therefore, the variables must be used in more than one translation unit.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>

N.	MISRA Definition	Messages in report file	Polyspace Implementation
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage	static storage class specifier should be used on internal linkage symbol XX.	
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization	Size of array 'XX' should be explicitly stated.	

Initialization

N.	MISRA Definition	Messages in report file	Polyspace Implementation
9.1	All automatic variables shall have been assigned a value before being used.		<p>Checked during code analysis.</p> <p>Violations displayed as Non-initialized variable results.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results. In Code Prover, you can also see a difference in results based on your choice for the option Verification level (-to). See “Check for Coding Standard Violations” on page 12-2.</p>
9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.	
9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	

Arithmetic Type Conversion

N.	MISRA Definition	Messages in report file	Polyspace Implementation
10.1	<p>The value of an expression of integer type shall not be implicitly converted to a different underlying type if:</p> <ul style="list-style-type: none"> • it is not a conversion to a wider integer type of the same signedness, or • the expression is complex, or • the expression is not constant and is a function argument, or • the expression is not constant and is a return expression 	<ul style="list-style-type: none"> • Implicit conversion of the expression of underlying type XX to the type XX that is not a wider integer type of the same signedness. • Implicit conversion of one of the binary operands whose underlying types are XX and XX • Implicit conversion of the binary right hand operand of underlying type XX to XX that is not an integer type. • Implicit conversion of the binary left hand operand of underlying type XX to XX that is not an integer type. • Implicit conversion of the binary right hand operand of underlying type XX to XX that is not a wider integer type of the same signedness or • Implicit conversion of the binary ? left hand operand of underlying type XX to XX, but it is a complex expression. • Implicit conversion of complex integer expression of underlying type XX to XX. • Implicit conversion of non-constant integer expression of underlying type XX in function return whose expected type is XX. • Implicit conversion of non-constant integer expression of underlying type XX as argument of function whose corresponding parameter type is XX. 	<p>ANSI C base types order (signed char, short, int, long) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1. The same interpretation is applied on the unsigned version of base types.</p> <p>An expression of bool or enum types has int as underlying type.</p> <p>Plain char may have signed or unsigned underlying type (depending on Polyspace target configuration or option setting).</p> <p>The underlying type of a simple expression of struct.bitfield is the base type used in the bitfield definition, the bitfield width is not token into account and it assumes that only signed unsigned int are used for bitfield (Rule 6.4).</p> <p>This rule violation is not produced on operations involving pointers.</p> <p>No violation reported when:</p> <ul style="list-style-type: none"> • The implicit conversion is a type widening, without change of signedness of integer • The expression is an argument expression or a return expression <p>No violation reported when the following are true:</p> <ul style="list-style-type: none"> • Implicit conversion applies to a constant expression and is a type widening, with a possible change of signedness of integer. • The conversion does not change the representation of

N.	MISRA Definition	Messages in report file	Polyspace Implementation
			<p>the constant value or the result of the operation.</p> <ul style="list-style-type: none"> The expression is an argument expression or a return expression or an operand expression of a non-bitwise operator. <p>Conversions of constants are not reported for these cases to avoid flagging too many violations. If the constant can be represented in both the original and converted type, the conversion is less of an issue.</p>
10.2	<p>The value of an expression of floating type shall not be implicitly converted to a different type if</p> <ul style="list-style-type: none"> it is not a conversion to a wider floating type, or the expression is complex, or the expression is a function argument, or the expression is a return expression 	<ul style="list-style-type: none"> Implicit conversion of the expression from XX to XX that is not a wider floating type. Implicit conversion of the binary ? right hand operand from XX to XX, but it is a complex expression. Implicit conversion of the binary ? right hand operand from XX to XX that is not a wider floating type or Implicit conversion of the binary ? left hand operand from XX to XX, but it is a complex expression. Implicit conversion of complex floating expression from XX to XX. Implicit conversion of floating expression of XX type in function return whose expected type is XX. Implicit conversion of floating expression of XX type as argument of function whose corresponding parameter type is XX. 	<p>ANSI C base types order (float, double) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1.</p> <p>No violation reported when:</p> <ul style="list-style-type: none"> The implicit conversion is a type widening The expression is an argument expression or a return expression.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression	Complex expression of underlying type XX may only be cast to narrower integer type of same signedness, however the destination type is XX.	<ul style="list-style-type: none"> • The rule checker raises a defect only if the result of a composite expression is cast to a different or wider essential type. For instance, in this example, a violation is shown in the first assignment to <code>i</code> but not the second. In the first assignment, a composite expression <code>i+1</code> is directly cast from a signed to an unsigned type. In the second assignment, the composite expression is first cast to the same type and then the result is cast to a different type. <pre>typedef int int32_T; typedef unsigned char uint8_T; ... int32_T i; i = (uint8_T)(i+1); /* Noncompliant */ i = (uint8_T)((int32_T)(i+1)); /* Compliant */</pre> • ANSI C base types order (signed char, short, int, long) defines that T1 is narrower than T2 if T2 is on the right hand of T1 or T1 = T2. The same methodology is applied on the unsigned version of base types. • An expression of bool or enum types has int as underlying type. • Plain char may have signed or unsigned underlying type (depending on target configuration or option setting). • The underlying type of a simple expression of struct.bitfield is the base type used in the bitfield definition, the bitfield width is not taken into account and it assumes that only signed,

N.	MISRA Definition	Messages in report file	Polyspace Implementation
			unsigned int are used for bitfield (Rule 6.4).
10.4	The value of a complex expression of float type may only be cast to narrower floating type	Complex expression of XX type may only be cast to narrower floating type, however the destination type is XX.	ANSI C base types order (float, double) defines that T1 is narrower than T2 if T2 is on the right hand of T1 or T2 = T1.
10.5	If the bitwise operator ~ and << are applied to an operand of underlying type <i>unsigned char</i> or <i>unsigned short</i> , the result shall be immediately cast to the underlying type of the operand	Bitwise [<< ~] is applied to the operand of underlying type [unsigned char unsigned short], the result shall be immediately cast to the underlying type.	
10.6	The "U" suffix shall be applied to all constants of <i>unsigned</i> types	No explicit 'U' suffix on constants of an unsigned type.	Warning when the type determined from the value and the base (octal, decimal or hexadecimal) is unsigned and there is no suffix u or U. For example, when the size of the <code>int</code> and <code>long int</code> data types is 32 bits, the coding rule checker will report a violation of rule 10.6 for the following line: <code>int a = 2147483648;</code> There is a difference between decimal and hexadecimal constants when <code>int</code> and <code>long int</code> are not the same size.

Pointer Type Conversion

N.	MISRA Definition	Messages in report file	Polyspace Implementation
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type	Conversion shall not be performed between a pointer to a function and any type other than an integral type.	Casts and implicit conversions involving a function pointer. Casts or implicit conversions from NULL or (void*)0 do not give any warning.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.	There is also a warning on qualifier loss This rule maps to ISO/IEC TS 17961 ID <code>alignconv</code> .

N.	MISRA Definition	Messages in report file	Polyspace Implementation
11.3	A cast should not be performed between a pointer type and an integral type	A cast should not be performed between a pointer type and an integral type.	Exception on zero constant. Extended to all conversions This rule maps to ISO/IEC TS 17961 ID alignconv .
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.	A cast should not be performed between a pointer to object type and a different pointer to object type.	
11.5	A cast shall not be performed that removes any <i>const</i> or <i>volatile</i> qualification from the type addressed by a pointer	A cast shall not be performed that removes any <i>const</i> or <i>volatile</i> qualification from the type addressed by a pointer	Extended to all conversions

Expressions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
12.1	Limited dependence should be placed on C's operator precedence rules in expressions	Limited dependence should be placed on C's operator precedence rules in expressions	
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits.	<ul style="list-style-type: none"> The value of '<i>sym</i>' depends on the order of evaluation. The value of volatile '<i>sym</i>' depends on the order of evaluation because of multiple accesses. 	<p>Rule 12.2 check assumes that no assignment in expressions that yield a Boolean values (rule 13.1).</p> <p>The expression is a simple expression of symbols. <code>i = i++;</code> is a violation, but <code>tab[2] = tab[2]++;</code> is not a violation.</p>
12.3	The <code>sizeof</code> operator should not be used on expressions that contain side effects.	The <code>sizeof</code> operator should not be used on expressions that contain side effects.	No warning on volatile accesses
12.4	The right hand operand of a logical <code>&&</code> or <code> </code> operator shall not contain side effects.	The right hand operand of a logical <code>&&</code> or <code> </code> operator shall not contain side effects.	No warning on volatile accesses
12.5	The operands of a logical <code>&&</code> or <code> </code> shall be primary-expressions.	<ul style="list-style-type: none"> operand of logical <code>&&</code> is not a primary expression operand of logical <code> </code> is not a primary expression The operands of a logical <code>&&</code> or <code> </code> shall be primary-expressions. 	<p>During preprocessing, violations of this rule are detected on the expressions in <code>#if</code> directives.</p> <p>Allowed exception on associatively (<code>a && b && c</code>), (<code>a b c</code>).</p>

N.	MISRA Definition	Messages in report file	Polyspace Implementation
12.6	Operands of logical operators (&&, and !) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (&&, or !).	<ul style="list-style-type: none"> • Operand of '!' logical operator should be effectively Boolean. • Left operand of '%s' logical operator should be effectively Boolean. • Right operand of '%s' logical operator should be effectively Boolean. • %s operand of '%s' is effectively Boolean. Boolean should not be used as operands to operators other than '&&', ' ', '!', '=', '==', '!=', and '?:'. 	<p>The operand of a logical operator should be a Boolean data type. Although the C standard does not explicitly define the Boolean data type, the standard implicitly assumes the use of the Boolean data type.</p> <p>Some operators may return Boolean-like expressions, for example, <code>(var == 0)</code>.</p> <p>Consider the following code:</p> <pre>unsigned char flag; if (!flag)</pre> <p>The rule checker reports a violation of rule 12.6:</p> <pre>Operand of '!' logical operator should be effectively Boolean.</pre> <p>The operand <code>flag</code> is not a Boolean but an <code>unsigned char</code>.</p> <p>To be compliant with rule 12.6, the code must be rewritten either as</p> <pre>if (!(flag != 0)) or if (flag == 0)</pre> <p>The use of the option <code>-boolean-types</code> may increase or decrease the number of warnings generated.</p>
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed	<ul style="list-style-type: none"> • [<code>~/Left Shift/Right shift/&</code>] operator applied on an expression whose underlying type is signed. • Bitwise <code>~</code> on operand of signed underlying type <code>XX</code>. • Bitwise [<code><< >></code>] on left hand operand of signed underlying type <code>XX</code>. • Bitwise [<code>& ^</code>] on two operands of <code>s</code> 	<p>The underlying type for an integer is signed when:</p> <ul style="list-style-type: none"> • it does not have a <code>u</code> or <code>U</code> suffix • it is small enough to fit into a 64 bits signed number

N.	MISRA Definition	Messages in report file	Polyspace Implementation
12.8	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.	<ul style="list-style-type: none"> • shift amount is negative • shift amount is bigger than 64 • Bitwise [<< >>] count out of range [0 ..X] (width of the underlying type XX of the left hand operand - 1).. 	<p>The numbers that are manipulated in preprocessing directives are 64 bits wide so that valid shift range is between 0 and 63</p> <p>Check is also extended onto bitfields with the field width or the width of the base type when it is within a complex expression</p>
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	<ul style="list-style-type: none"> • Unary - on operand of unsigned underlying type XX. • Minus operator applied to an expression whose underlying type is unsigned 	<p>The underlying type for an integer is signed when:</p> <ul style="list-style-type: none"> • it does not have a u or U suffix • it is small enough to fit into a 64 bits signed number
12.10	The comma operator shall not be used.	The comma operator shall not be used.	
12.11	Evaluation of constant unsigned expression should not lead to wraparound.	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	
12.12	The underlying bit representations of floating-point values shall not be used.	The underlying bit representations of floating-point values shall not be used.	<p>Warning when:</p> <ul style="list-style-type: none"> • A float pointer is cast as a pointer to another data type. Casting a float pointer as a pointer to void does not generate a warning. • A float is packed with another data type. For example: <pre> union { float f; int i; } ... </pre>
12.13	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression	Warning when ++ or -- operators are not used alone.

Control Statement Expressions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
13.1	Assignment operators shall not be used in expressions that yield Boolean values.	Assignment operators shall not be used in expressions that yield Boolean values.	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	No warning is given on integer constants. Example: if (2) The use of the option <code>-boolean-types</code> may increase or decrease the number of warnings generated.
13.3	Floating-point expressions shall not be tested for equality or inequality.	Floating-point expressions shall not be tested for equality or inequality.	Warning on direct tests only.
13.4	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type	If <i>for</i> index is a variable symbol, checked that it is not a float.
13.5	The three expressions of a <i>for</i> statement shall be concerned only with loop control	<ul style="list-style-type: none"> • 1st expression should be an assignment. • Bad type for loop counter (XX). • 2nd expression should be a comparison. • 2nd expression should be a comparison with loop counter (XX). • 3rd expression should be an assignment of loop counter (XX). • 3rd expression: assigned variable should be the loop counter (XX). • The following kinds of <i>for</i> loops are allowed: <ul style="list-style-type: none"> (a) all three expressions shall be present; (b) the 2nd and 3rd expressions shall be present with prior initialization of the loop counter; (c) all three expressions shall be empty for a deliberate infinite loop. 	Checked if the <i>for</i> loop index (V) is a variable symbol; checked if V is the last assigned variable in the first expression (if present). Checked if, in first expression, if present, is assignment of V; checked if in 2nd expression, if present, must be a comparison of V; Checked if in 3rd expression, if present, must be an assignment of V.
13.6	Numeric variables being used within a <i>for</i> loop for iteration counting should not be modified in the body of the loop.	Numeric variables being used within a <i>for</i> loop for iteration counting should not be modified in the body of the loop.	Detect only direct assignments if the <i>for</i> loop index is known and if it is a variable symbol.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
13.7	Boolean operations whose results are invariant shall not be permitted	<ul style="list-style-type: none"> • Boolean operations whose results are invariant shall not be permitted. Expression is always true. • Boolean operations whose results are invariant shall not be permitted. Expression is always false. • Boolean operations whose results are invariant shall not be permitted. 	<p>During compilation, check comparisons with at least one constant operand.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p> <ul style="list-style-type: none"> • Bug Finder flags some violations of this rule through the <code>Dead code</code> and <code>Useless if</code> checkers. • Code Prover does not use gray code to flag violations of this rule. <p>In Code Prover, you can also see a difference in results based on your choice for the option <code>Verification level (-to)</code>. See “Check for Coding Standard Violations” on page 12-2..</p> <p>The rule violation appears when you check whether an <code>enum</code> variable value lies between its lower and upper bound. The violation appears even if you increment or decrement the variable outside its bounds, for instance, in this <code>for</code> loop condition:</p> <pre>enum ec {RED, BLUE, GREEN} col; for(col=RED; col<=GREEN; col++) {}</pre> <p>An <code>enum</code> 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 <code>enum</code> to an integer before the comparison, for instance:</p> <pre>enum ec {RED, BLUE, GREEN} col; for(col=RED; (int)col<=GREEN; col++) {}</pre>

Control Flow

N.	MISRA Definition	Messages in report file	Polyspace Implementation
14.1	There shall be no unreachable code.	There shall be no unreachable code.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
14.2	All non-null statements shall either have at least one side effect however executed, or cause control flow to change	All non-null statements shall either: <ul style="list-style-type: none"> • have at least one side effect however executed, or • cause control flow to change 	
14.3	Before preprocessing, a null statement shall occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.	A null statement shall appear on a line by itself	We assume that a ';' is a null statement when it is the first character on a line (excluding comments). The rule is violated when: <ul style="list-style-type: none"> • there are some comments before it on the same line. • there is a comment immediately after it • there is something else than a comment after the ';' on the same line.
14.4	The <i>goto</i> statement shall not be used.	The goto statement shall not be used.	
14.5	The <i>continue</i> statement shall not be used.	The continue statement shall not be used.	
14.6	For any iteration statement there shall be at most one <i>break</i> statement used for loop termination	For any iteration statement there shall be at most one break statement used for loop termination	
14.7	A function shall have a single point of exit at the end of the function	A function shall have a single point of exit at the end of the function	
14.8	The statement forming the body of a <i>switch</i> , <i>while</i> , <i>do while</i> or <i>for</i> statement shall be a compound statement	<ul style="list-style-type: none"> • The body of a do while statement shall be a compound statement. • The body of a for statement shall be a compound statement. • The body of a switch statement shall be a compound statement 	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
14.9	An <i>if (expression)</i> construct shall be followed by a compound statement. The <i>else</i> keyword shall be followed by either a compound statement, or another <i>if</i> statement	<ul style="list-style-type: none"> • An <i>if (expression)</i> construct shall be followed by a compound statement. • The <i>else</i> keyword shall be followed by either a compound statement, or another <i>if</i> statement 	
14.10	All <i>if else if</i> constructs should contain a final <i>else</i> clause.	All <i>if else if</i> constructs should contain a final <i>else</i> clause.	

Switch Statements

N.	MISRA Definition	Messages in report file	Polyspace Implementation
15.0	The MISRA C switch syntax shall be used.	switch statements syntax normative restrictions.	<p>Warning on declarations or any statements before the first switch case.</p> <p>Warning on label or jump statements in the body of switch cases.</p> <p>On the following example, the rule is displayed in the log file at line 3:</p> <pre> 1 ... 2 switch(index) { 3 var = var + 1; // RULE 15.0 // violated 4case 1: ... </pre> <p>The code between switch statement and first case is checked as dead code by Polyspace. It follows ANSI standard behavior.</p> <p>This rule is not considered as a required rule in the MISRA C:2004 rules for generated code. In generated code, if you find a violation of rule 15.0 that does not simultaneously violate a later rule in this group, justify the violation with appropriate comments.</p>

N.	MISRA Definition	Messages in report file	Polyspace Implementation
15.1	A switch label shall only be used when the most closely-enclosing compound statement is the body of a <i>switch</i> statement	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	
15.2	An unconditional <i>break</i> statement shall terminate every non-empty switch clause	An unconditional break statement shall terminate every non-empty switch clause	Warning for each non-compliant case clause.
15.3	The final clause of a <i>switch</i> statement shall be the <i>default</i> clause	The final clause of a switch statement shall be the default clause	
15.4	A <i>switch</i> expression should not represent a value that is effectively Boolean	A switch expression should not represent a value that is effectively Boolean	The use of the option <code>-boolean-types</code> may increase the number of warnings generated.
15.5	Every <i>switch</i> statement shall have at least one <i>case</i> clause	Every switch statement shall have at least one case clause	

Functions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
16.1	Functions shall not be defined with variable numbers of arguments.	Function XX should not be defined as varargs.	
16.2	Functions shall not call themselves, either directly or indirectly.	Function %s should not call itself.	The checker reports each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported. You can calculate the total number of recursion cycles using the code complexity metric <code>Number of Recursions</code> .
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.	Identifiers shall be given for all of the parameters in a function prototype declaration.	Assumes Rule 8.6 is not violated.
16.4	The identifiers used in the declaration and definition of a function shall be identical.	The identifiers used in the declaration and definition of a function shall be identical.	Assumes that rules 8.8 , 8.1 and 16.3 are not violated. All occurrences are detected.
16.5	Functions with no parameters shall be declared with parameter type <i>void</i> .	Functions with no parameters shall be declared with parameter type void.	Definitions are also checked.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
16.6	The number of arguments passed to a function shall match the number of parameters.	<ul style="list-style-type: none"> • Too many arguments to XX. • Insufficient number of arguments to XX. 	Assumes that rule 8.1 is not violated. This rule maps to ISO/IEC TS 17961 ID argcomp .
16.7	A pointer parameter in a function prototype should be declared as pointer to <code>const</code> if the pointer is not used to modify the addressed object.	Pointer parameter in a function prototype should be declared as pointer to <code>const</code> if the pointer is not used to modify the addressed object.	Warning if a non- <code>const</code> 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 <code>const</code> 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.
16.9	A function identifier shall only be used with either a preceding <code>&</code> , or with a parenthesized parameter list, which may be empty.	Function identifier XX should be preceded by a <code>&</code> 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 <code>memcpy</code> , <code>memset</code> , <code>memmove</code> , <code>strcpy</code> , <code>strncpy</code> , <code>strcat</code> , <code>strncat</code> 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.	
17.3	<code>></code> , <code>>=</code> , <code><</code> , <code><=</code> shall not be applied to pointer types except where they point to the same array.	<code>></code> , <code>>=</code> , <code><</code> , <code><=</code> shall not be applied to pointer types except where they point to the same array.	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
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: <ul style="list-style-type: none"> Operations on pointers. ($p+I$, $I+p$, and $p-I$, where p is a pointer and I an integer). Array indexing on nonarray pointers.
17.5	A type should not contain more than 2 levels of pointer indirection	A type should not contain more than 2 levels of pointer indirection	
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.	Pointer to a parameter is an illegal return value. Pointer to a local is an illegal return value.	Warning when assigning address to a global variable, returning a local variable address, or returning a parameter address. This rule maps to ISO/IEC TS 17961 ID accfree .

Structures and Unions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
18.1	All structure or union types shall be complete at the end of a translation unit.	All structure or union types shall be complete at the end of a translation unit.	Warning for all incomplete declarations of structs or unions.
18.2	An object shall not be assigned to an overlapping object.	<ul style="list-style-type: none"> An object shall not be assigned to an overlapping object. Destination and source of XX overlap, the behavior is undefined. 	
18.4	Unions shall not be used	Unions shall not be used.	

Preprocessing Directives

N.	MISRA Definition	Messages in report file	Polyspace Implementation
19.1	<code>#include</code> statements in a file shall only be preceded by other preprocessors directives or comments	<code>#include</code> statements in a file shall only be preceded by other preprocessors directives or comments	A message is displayed when a <code>#include</code> directive is preceded by other things than preprocessor directives, comments, spaces or "new lines".
19.2	Nonstandard characters should not occur in header file names in <code>#include</code> directives	<ul style="list-style-type: none"> A message is displayed on characters <code>'</code>, <code>"</code> or <code>/*</code> between <code><</code> and <code>></code> in <code>#include <filename></code> A message is displayed on characters <code>'</code>, or <code>/*</code> between <code>"</code> and <code>"</code> in <code>#include "filename"</code> 	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
19.3	The <i>#include</i> directive shall be followed by either a <filename> or "filename" sequence.	<ul style="list-style-type: none"> • '#include' expects "FILENAME" or <FILENAME> • '#include_next' expects "FILENAME" or <FILENAME> 	
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.	Macro '<name>' does not expand to a compliant construct.	<p>We assume that a macro definition does not violate this rule when it expands to:</p> <ul style="list-style-type: none"> • a braced construct (not necessarily an initializer) • a parenthesized construct (not necessarily an expression) • a number • a character constant • a string constant (can be the result of the concatenation of string field arguments and literal strings) • the following keywords: typedef, extern, static, auto, register, const, volatile, <code>__asm__</code> and <code>__inline__</code> • a do-while-zero construct
19.5	Macros shall not be <i>#defined</i> and <i>#undefd</i> within a block.	<ul style="list-style-type: none"> • Macros shall not be <i>#define'd</i> within a block. • Macros shall not be <i>#undef'd</i> within a block. 	
19.6	<i>#undef</i> shall not be used.	<i>#undef</i> shall not be used.	
19.7	A function should be used in preference to a function like-macro.	A function should be used in preference to a function like-macro	Message on all function-like macro definitions.
19.8	A function-like macro shall not be invoked without all of its arguments	<ul style="list-style-type: none"> • arguments given to macro '<name>' • macro '<name>' used without args. • macro '<name>' used with just one arg. • macro '<name>' used with too many (<number>) args. 	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
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)
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.	Parameter instance shall be enclosed in parentheses.	<p>If x is a macro parameter, the following instances of x as an operand of the # and ## operators do not generate a warning: #x, ##x, and x##. Otherwise, parentheses are required around x.</p> <p>The software does not generate a warning if a parameter is reused as an argument of a function or function-like macro. For example, consider a parameter x. The software does not generate a warning if x appears as (x) or (x, or ,x) or ,x,.</p>
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.	'<name>' is not defined.	
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.	More than one occurrence of the # or ## preprocessor operators.	
19.13	The # and ## preprocessor operators should not be used	Message on definitions of macros using # or ## operators	
19.14	The defined preprocessor operator shall only be used in one of the two standard forms.	'defined' without an identifier.	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.	Precautions shall be taken in order to prevent multiple inclusions.	When a header file is formatted as, <pre>#ifndef <control macro> #define <control macro> <contents> #endif</pre> or, <pre>#ifndef <control macro> #error ... #else #define <control macro> <contents> #endif</pre> it is assumed that precautions have been taken to prevent multiple inclusions. Otherwise, a violation of this MISRA rule is detected.
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	directive is not syntactically meaningful.	
19.17	All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> or <code>#ifdef</code> directive to which they are related.	<ul style="list-style-type: none"> • <code>'#elif'</code> not within a conditional. • <code>'#else'</code> not within a conditional. • <code>'#elif'</code> not within a conditional. • <code>'#endif'</code> not within a conditional. • unbalanced <code>'#endif'</code>. • unterminated <code>'#if'</code> conditional. • unterminated <code>'#ifdef'</code> conditional. • unterminated <code>'#ifndef'</code> conditional. 	

Standard Libraries

N.	MISRA Definition	Messages in report file	Polyspace Implementation
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be redefined. • The macro '<code><name></code>' shall not be undefined. 	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
20.2	The names of standard library macros, objects and functions shall not be reused.	Identifier XX should not be used.	<p>In case a macro whose name corresponds to a standard library macro, object or function is defined, the rule that is detected as violated is 20.1.</p> <p>Tentative definitions are considered as definitions. For objects with file scope, tentative definitions are declarations that:</p> <ul style="list-style-type: none"> • Do not have initializers. • Do not have storage class specifiers, or have the <code>static</code> specifier
20.3	The validity of values passed to library functions shall be checked.	Validity of values passed to library functions shall be checked	<p>Warning for argument in library function call if the following are all true:</p> <ul style="list-style-type: none"> • Argument is a local variable • Local variable is not tested between last assignment and call to the library function • Library function is a common mathematical function • Corresponding parameter of the library function has a restricted input domain. <p>The library function can be one of the following : <code>sqrt</code>, <code>tan</code>, <code>pow</code>, <code>log</code>, <code>log10</code>, <code>fmod</code>, <code>acos</code>, <code>asin</code>, <code>acosh</code>, <code>atanh</code>, or <code>atan2</code>.</p> <p>Bug Finder and Code Prover check this rule differently. The analysis can produce different results.</p>
20.4	Dynamic heap memory allocation shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	In case the dynamic heap memory allocation functions are actually macros and the macro is expanded in the code, this rule is detected as violated. Assumes rule 20.2 is not violated.
20.5	The error indicator <code>errno</code> shall not be used	The error indicator <code>errno</code> shall not be used	Assumes that rule 20.2 is not violated

N.	MISRA Definition	Messages in report file	Polyspace Implementation
20.6	The macro <i>offsetof</i> , in library <code><stddef.h></code> , shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	Assumes that rule 20.2 is not violated
20.7	The <i>setjmp</i> macro and the <i>longjmp</i> function shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	In case the <i>longjmp</i> function is actually a macro and the macro is expanded in the code, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.8	The signal handling facilities of <code><signal.h></code> shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	In case some of the signal functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.9	The input/output library <code><stdio.h></code> shall not be used in production code.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	In case the input/output library functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.10	The library functions <i>atof</i> , <i>atoi</i> and <i>atoll</i> from library <code><stdlib.h></code> shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	In case the <i>atof</i> , <i>atoi</i> and <i>atoll</i> functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.11	The library functions <i>abort</i> , <i>exit</i> , <i>getenv</i> and <i>system</i> from library <code><stdlib.h></code> shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	In case the <i>abort</i> , <i>exit</i> , <i>getenv</i> and <i>system</i> functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.12	The time handling functions of library <code><time.h></code> shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	In case the time handling functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule 20.2 is not violated

Runtime Failures

N.	MISRA Definition	Messages in report file	Polyspace Implementation
21.1	Minimization of runtime failures shall be ensured by the use of at least one of: <ul style="list-style-type: none"> • static verification tools/ techniques; • dynamic verification tools/ techniques; • explicit coding of checks to handle runtime faults. 		Done by Polyspace. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results. In Code Prover, you can also see a difference in results based on your choice for the option <code>Verification level (-to)</code> . See "Check for Coding Standard Violations" on page 12-2..

Unsupported MISRA C:2004 and MISRA AC AGC Rules

The Polyspace coding rules checker does not check the following MISRA C:2004 coding rules. These rules cannot be enforced because they are outside the scope of Polyspace software. They may concern documentation, dynamic aspects, or functional aspects of MISRA rules. The **Additional Information** column describes the reason each rule is not checked.

Environment

Rule	Description	Additional Information
1.2 (Required)	No reliance shall be placed on undefined or unspecified behavior	Not statically checkable unless the data dynamic properties is taken into account
1.3 (Required)	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the language/compiler/assemblers conform.	It is a process rule method.
1.4 (Required)	The compiler/linker/Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.	To observe this rule, check your compiler documentation.
1.5 (Advisory)	Floating point implementations should comply with a defined floating point standard.	To observe this rule, check your compiler documentation.

Language Extensions

Rule	Description	Additional Information
2.4 (Advisory)	Sections of code should not be “commented out”	One way a tool can check this rule is to determine if the code compiles when commented out sections are uncommented. However, such checking can be expensive and inaccurate.

Documentation

Rule	Description	Additional Information
3.1 (Required)	All usage of implementation-defined behavior shall be documented.	To observe this rule, check your compiler documentation. Error detection is based on undefined behavior, according to choices made for implementation- defined constructions.
3.2 (Required)	The character set and the corresponding encoding shall be documented.	To observe this rule, check your compiler documentation.
3.3 (Advisory)	The implementation of integer division in the chosen compiler should be determined, documented and taken into account.	To observe this rule, check your compiler documentation.
3.5 (Required)	The implementation-defined behavior and packing of bitfields shall be documented if being relied upon.	To observe this rule, check your compiler documentation.
3.6 (Required)	All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.	To observe this rule, check your compiler documentation.

Structures and Unions

Rule	Description	Additional Information
18.3 (Required)	An area of memory shall not be reused for unrelated purposes.	"purpose" is functional design issue.

Polyspace MISRA C:2012 Checkers

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

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

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

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

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

There are additional subsets of MISRA C:2012 guidelines defined by Polyspace called Software Quality Objectives (SQO) that can have a direct or indirect impact on the precision of your results. When you set up checking, you can select these subsets. These subsets are defined in “Software Quality Objective Subsets (C:2012)” on page 12-18.

See Also

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

See Also

More About

- “Check for Coding Rule Violations” on page 5-14
- “MISRA C:2012 Directives and Rules”

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

Essential Types in MISRA C: 2012 Rules 10.x

MISRA C: 2012 rules 10.x classify data types in categories. The rules treat data types in the same category as essentially similar.

For instance, the data types `float`, `double` and `long double` are considered as essentially floating. Rule 10.1 states that the `%` operation must not have essentially floating operands. This statement implies that the operands cannot have one of these three data types: `float`, `double` and `long double`.

Categories of Essential Types

The essential types fall in these categories:

Essential type category	Standard types
Essentially Boolean	<code>bool</code> or <code>_Bool</code> (defined in <code>stdbool.h</code>) If you define a boolean type through a <code>typedef</code> , you must specify this type name before coding rules checking. For more information, see Effective boolean types (-boolean-types) ..
Essentially character	<code>char</code>
Essentially enum	named enum
Essentially signed	<code>signed char</code> , <code>signed short</code> , <code>signed int</code> , <code>signed long</code> , <code>signed long long</code>
Essentially unsigned	<code>unsigned char</code> , <code>unsigned short</code> , <code>unsigned int</code> , <code>unsigned long</code> , <code>unsigned long long</code>
Essentially floating	<code>float</code> , <code>double</code> , <code>long double</code>

How MISRA C: 2012 Uses Essential Types

These rules use essential types in their statements:

- MISRA C: 2012 Rule 10.1: Operands shall not be of an inappropriate essential type.

For instance, the right operand of the `<<` or `>>` operator must be essentially unsigned. Otherwise, negative values can cause undefined behavior.

- MISRA C: 2012 Rule 10.2: Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.

For instance, the type `char` does not represent numeric values. Do not use a variable of this type in addition and subtraction operations.

- MISRA C: 2012 Rule 10.3: The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.

For instance, do not assign a variable of data type `double` to a variable with the narrower data type `float`.

- MISRA C: 2012 Rule 10.4: Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.

For instance, do not perform an addition operation with a signed `int` operand, which belongs to the essentially signed category, and an unsigned `int` operand, which belongs to the essentially unsigned category.

- MISRA C: 2012 Rule 10.5: The value of an expression should not be cast to an inappropriate essential type.

For instance, do not perform a cast between essentially floating types and essentially character types.

- MISRA C: 2012 Rule 10.6: The value of a composite expression shall not be assigned to an object with wider essential type.

For instance, if a multiplication, binary addition or bitwise operation involves unsigned `char` operands, do not assign the result to a variable having the wider type unsigned `int`.

- MISRA C: 2012 Rule 10.7: If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.

For instance, if one operand of an addition operation is a composite expression with two unsigned `char` operands, the other operand must not have the wider type unsigned `int`.

See Also

More About

- “Check for Coding Standard Violations” on page 12-2
- “MISRA C:2012 Directives and Rules”

Unsupported MISRA C:2012 Guidelines

The Polyspace coding rules checker does not check the following MISRA C:2012 directives. These directives are not checked either in Bug Finder or Code Prover. These directives cannot be enforced because they are outside the scope of Polyspace software. These guidelines concern documentation, dynamic aspects, or functional aspects of MISRA rules.

For the list of supported rules and directives, see “MISRA C:2012 Directives and Rules”.

Number	Category	AGC Category	Definition
Directive 3.1	Required	Required	All code shall be traceable to documented requirements
Directive 4.2	Advisory	Advisory	All usage of assembly language should be documented
Directive 4.4	Advisory	Advisory	Sections of code should not be “commented out”

See Also

More About

- “MISRA C:2012 Directives and Rules”

Polyspace MISRA C++ Checkers

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

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

There are subsets of MISRA C++ coding rules that can have a direct or indirect impact on the selectivity (reliability percentage) of your results. When you set up rule checking, you can select these subsets directly. These subsets are defined in “Software Quality Objective Subsets (C++)” on page 12-21.

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

See Also

More About

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

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

Unsupported MISRA C++ Coding Rules

In this section...
“Language Independent Issues” on page 13-39
“General” on page 13-40
“Lexical Conventions” on page 13-40
“Expressions” on page 13-40
“Declarations” on page 13-41
“Classes” on page 13-41
“Templates” on page 13-41
“Exception Handling” on page 13-41
“Library Introduction” on page 13-42

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

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

Language Independent Issues

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

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

General

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

Lexical Conventions

N.	Category	MISRA Definition	Additional Information
2-2-1	Document	The character set and the corresponding encoding shall be documented.	To observe this rule, check your compiler documentation.
2-7-2	Required	Sections of code shall not be "commented out" using C-style comments.	One way a tool can check this rule is to determine if the code compiles when commented out sections are uncommented. However, such checking can be expensive and inaccurate.
2-7-3	Advisory	Sections of code should not be "commented out" using C++ comments.	One way a tool can check this rule is to determine if the code compiles when commented out sections are uncommented. However, such checking can be expensive and inaccurate.

Expressions

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

Declarations

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

Classes

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

Templates

N.		MISRA Definition	Additional Information
14-5-1	Required	A non-member generic function shall only be declared in a namespace that is not an associated namespace.	
14-7-1	Required	All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.	
14-7-2	Required	For any given template specialization, an explicit instantiation of the template with the template-arguments used in the specialization shall not render the program ill-formed.	

Exception Handling

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

N.	Category	MISRA Definition	Additional Information
15-3-4	Required	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	

Library Introduction

N.	Category	MISRA Definition	Additional Information
17-0-3	Required	The names of standard library functions shall not be overridden.	
17-0-4	Required	All library code shall conform to MISRA C++.	To observe this rule, check your compiler documentation.

See Also

More About

- “MISRA C++:2008 Rules”

Polyspace JSF C++ Checkers

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

4

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

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

See Also

More About

- “Check for Coding Standard Violations” on page 12-2
- “JSF C++ Coding Rules” on page 13-44

4. JSF and Joint Strike Fighter are Lockheed Martin.

JSF 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: <i><function name></i> has <i><num></i> logical source lines of code.
3	All functions shall have a cyclomatic complexity number of 20 or less.	Message in report file: <i><function name></i> has cyclomatic complexity number equal to <i><num></i> .

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: <i><digraph></i> . Reports the digraph. If the rule level is set to warning, the digraph will be allowed even if it is not supported in <code>-compiler iso</code> .
13	Multi-byte characters and wide string literals will not be used.	Report <code>L'c'</code> , <code>L"string"</code> , and use of <code>wchar_t</code> .
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 <code>errno</code> shall not be used.	<code>errno</code> should not be used as a macro or a global with external "C" linkage.
18	The macro <code>offsetof</code> , in library <code><stddef.h></code> , shall not be used.	<code>offsetof</code> should not be used as a macro or a global with external "C" linkage.

N.	JSF++ Definition	Polyspace Implementation
19	<locale.h> and the <code>setlocale</code> function shall not be used.	<code>setlocale</code> and <code>localeconv</code> should not be used as a macro or a global with external "C" linkage.
20	The <code>setjmp</code> macro and the <code>longjmp</code> function shall not be used.	<code>setjmp</code> and <code>longjmp</code> should not be used as a macro or a global with external "C" linkage.
21	The signal handling facilities of <signal.h> shall not be used.	<code>signal</code> and <code>raise</code> should not be used as a macro or a global with external "C" linkage.
22	The input/output library <stdio.h> shall not be used.	all standard functions of <stdio.h> should not be used as a macro or a global with external "C" linkage.
23	The library functions <code>atof</code> , <code>atoi</code> and <code>atol</code> from library <stdlib.h> shall not be used.	<code>atof</code> , <code>atoi</code> and <code>atol</code> should not be used as a macro or a global with external "C" linkage.
24	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> from library <stdlib.h> shall not be used.	<code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> should not be used as a macro or a global with external "C" linkage.
25	The time handling functions of library <time.h> shall not be used.	<code>clock</code> , <code>difftime</code> , <code>mktime</code> , <code>asctime</code> , <code>ctime</code> , <code>gmtime</code> , <code>localtime</code> and <code>strftime</code> should not be used as a macro or a global with external "C" linkage.

Pre-Processing Directives

N.	JSF++ Definition	Polyspace Implementation
26	Only the following preprocessor directives shall be used: <code>#ifndef</code> , <code>#define</code> , <code>#endif</code> , <code>#include</code> .	
27	<code>#ifndef</code> , <code>#define</code> and <code>#endif</code> will be used to prevent multiple inclusions of the same header file. Other techniques to prevent the multiple inclusions of header files will not be used.	Detects the patterns <code>#if !defined</code> , <code>#pragma once</code> , <code>#ifdef</code> , and missing <code>#define</code> .
28	The <code>#ifndef</code> and <code>#endif</code> preprocessor directives will only be used as defined in AV Rule 27 to prevent multiple inclusions of the same header file.	Detects any use that does not comply with AV Rule 27. Assuming 35/27 is not violated, reports only <code>#ifndef</code> .
29	The <code>#define</code> preprocessor directive shall not be used to create inline macros. Inline functions shall be used instead.	Rule is split into two parts: the definition of a macro function (29.def) and the call of a macrofunction (29.use). Messages in report file: <ul style="list-style-type: none"> • 29.1 : The <code>#define</code> preprocessor directive shall not be used to create inline macros. • 29.2 : Inline functions shall be used instead of inline macros.

N.	JSF++ Definition	Polyspace Implementation
30	The <code>#define</code> preprocessor directive shall not be used to define constant values. Instead, the <code>const</code> qualifier shall be applied to variable declarations to specify constant values.	Reports <code>#define</code> of simple constants.
31	The <code>#define</code> preprocessor directive will only be used as part of the technique to prevent multiple inclusions of the same header file.	Detects use of <code>#define</code> that are not used to guard for multiple inclusion, assuming that rules 35 and 27 are not violated.
32	The <code>#include</code> preprocessor directive will only be used to include header (*.h) files.	

Header Files

N.	JSF++ Definition	Polyspace Implementation
33	The <code>#include</code> directive shall use the <code><filename.h></code> notation to include header files.	
35	A header file will contain a mechanism that prevents multiple inclusions of itself.	
39	Header files (*.h) will not contain non-const variable definitions or function definitions.	Reports definitions of global variables / function in header.

Style

N.	JSF++ Definition	Polyspace Implementation
40	Every implementation file shall include the header files that uniquely define the inline functions, types, and templates used.	Reports when type, template, or inline function is defined in source file. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
41	Source lines will be kept to a length of 120 characters or less.	
42	Each expression-statement will be on a separate line.	Reports when two consecutive expression statements are on the same line (unless the statements are part of a macro definition).
43	Tabs should be avoided.	
44	All indentations will be at least two spaces and be consistent within the same source file.	Reports when a statement indentation is not at least two spaces more than the statement containing it. Does not report bad indentation between opening braces following <code>if/else</code> , <code>do/while</code> , <code>for</code> , and <code>while</code> statements. NB: in final release it will accept any indentation
46	User-specified identifiers (internal and external) will not rely on significance of more than 64 characters.	
47	Identifiers will not begin with the underscore character <code>'_'</code> .	

N.	JSF++ Definition	Polyspace Implementation
48	Identifiers will not differ by: <ul style="list-style-type: none"> • Only a mixture of case • The presence/absence of the underscore character • The interchange of the letter 'O'; with the number '0' or the letter 'D' • The interchange of the letter 'I'; with the number '1' or the letter 'l' • The interchange of the letter 'S' with the number '5' • The interchange of the letter 'Z' with the number 2 • The interchange of the letter 'n' with the letter 'h' 	Checked regardless of scope. Not checked between macros and other identifiers. Messages in report file: <ul style="list-style-type: none"> • Identifier <i>Idf1 (file1.cpp line l1 column c1)</i> and <i>Idf2 (file2.cpp line l2 column c2)</i> only differ by the presence/absence of the underscore character. • Identifier <i>Idf1 (file1.cpp line l1 column c1)</i> and <i>Idf2 (file2.cpp line l2 column c2)</i> only differ by a mixture of case. • Identifier <i>Idf1 (file1.cpp line l1 column c1)</i> and <i>Idf2 (file2.cpp line l2 column c2)</i> only differ by letter 0, with the number 0.
50	The first word of the name of a class, structure, namespace, enumeration, or type created with typedef will begin with an uppercase letter. All others letters will be lowercase.	Messages in report file: <ul style="list-style-type: none"> • The first word of the name of a class will begin with an uppercase letter. • The first word of the namespace of a class will begin with an uppercase letter. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
51	All letters contained in function and variables names will be composed entirely of lowercase letters.	Messages in report file: <ul style="list-style-type: none"> • All letters contained in variable names will be composed entirely of lowercase letters. • All letters contained in function names will be composed entirely of lowercase letters.
52	Identifiers for constant and enumerator values shall be lowercase.	Messages in report file: <ul style="list-style-type: none"> • Identifier for enumerator value shall be lowercase. • Identifier for template constant parameter shall be lowercase.
53	Header files will always have file name extension of ".h".	.H is allowed if you set the option -dos.
53.1	The following character sequences shall not appear in header file names: ' , \, /*, //, or ".	
54	Implementation files will always have a file name extension of ".cpp".	Not case sensitive if you set the option -dos.
57	The public, protected, and private sections of a class will be declared in that order.	

N.	JSF++ Definition	Polyspace Implementation
58	When declaring and defining functions with more than two parameters, the leading parenthesis and the first argument will be written on the same line as the function name. Each additional argument will be written on a separate line (with the closing parenthesis directly after the last argument).	Detects that two parameters are not on the same line, The first parameter should be on the same line as function name. Does not check for the closing parenthesis.
59	The statements forming the body of an if, else if, else, while, do ... while or for statement shall always be enclosed in braces, even if the braces form an empty block.	<p>Messages in report file:</p> <ul style="list-style-type: none"> • The statements forming the body of an if statement shall always be enclosed in braces. • The statements forming the body of an else statement shall always be enclosed in braces. • The statements forming the body of a while statement shall always be enclosed in braces. • The statements forming the body of a do ... while statement shall always be enclosed in braces. • The statements forming the body of a for statement shall always be enclosed in braces.
60	Braces ("{}") which enclose a block will be placed in the same column, on separate lines directly before and after the block.	Detects that statement-block braces should be in the same columns.
61	Braces ("{}") which enclose a block will have nothing else on the line except comments.	
62	The dereference operator '*' and the address-of operator '&' will be directly connected with the type-specifier.	Reports when there is a space between type and "*" "&" for variables, parameters and fields declaration.
63	Spaces will not be used around '.' or '->', nor between unary operators and operands.	<p>Reports when the following characters are not directly connected to a white space:</p> <ul style="list-style-type: none"> • . • -> • ! • ~ • - • ++ • — <p>Note that a violation will be reported for "." used in float/double definition.</p>

Classes

N.	JSF++ Definition	Polyspace Implementation
67	Public and protected data should only be used in structs - not classes.	
68	Unneeded implicitly generated member functions shall be explicitly disallowed.	Reports when default constructor, assignment operator, copy constructor or destructor is not declared.
71.1	A class's virtual functions shall not be invoked from its destructor or any of its constructors.	Reports when a constructor or destructor directly calls a virtual function.
74	Initialization of nonstatic class members will be performed through the member initialization list rather than through assignment in the body of a constructor.	All data should be initialized in the initialization list except for array. Does not report that an assignment exists in ctor body. Message in report file: Initialization of nonstatic class members "<field>" will be performed through the member initialization list.
75	Members of the initialization list shall be listed in the order in which they are declared in the class.	
76	A copy constructor and an assignment operator shall be declared for classes that contain pointers to data items or nontrivial destructors.	Messages in report file: <ul style="list-style-type: none"> • no copy constructor and no copy assign • no copy constructor • no copy assign Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
77.1	The definition of a member function shall not contain default arguments that produce a signature identical to that of the implicitly-declared copy constructor for the corresponding class/structure.	Does not report when an explicit copy constructor exists.
78	All base classes with a virtual function shall define a virtual destructor.	
79	All resources acquired by a class shall be released by the class's destructor.	Reports when the number of "new" called in a constructor is greater than the number of "delete" called in its destructor. Note A violation is raised even if "new" is done in a "if/else".

N.	JSF++ Definition	Polyspace Implementation
81	The assignment operator shall handle self-assignment correctly	<p>Reports when copy assignment body does not begin with "if (this != arg)"</p> <p>A violation is not raised if an empty else statement follows the if, or the body contains only a return statement.</p> <p>A violation is raised when the if statement is followed by a statement other than the return statement.</p>
82	An assignment operator shall return a reference to <code>*this</code> .	<p>The following operators should return <code>*this</code> on method, and <code>*first_arg</code> on plain function:</p> <ul style="list-style-type: none"> • operator= • operator+= • operator-= • operator*= • operator >>= • operator <<= • operator /= • operator %= • operator = • operator &= • operator ^= • Prefix operator++ • Prefix operator-- <p>Does not report when no return exists.</p> <p>No special message if type does not match.</p> <p>Messages in report file:</p> <ul style="list-style-type: none"> • An assignment operator shall return a reference to <code>*this</code>. • An assignment operator shall return a reference to its first arg.
83	An assignment operator shall assign all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).	Reports when a copy assignment does not assign all data members. In a derived class, it also reports when a copy assignment does not call inherited copy assignments.

N.	JSF++ Definition	Polyspace Implementation
88	Multiple inheritance shall only be allowed in the following restricted form: n interfaces plus m private implementations, plus at most one protected implementation.	Messages in report file: <ul style="list-style-type: none"> • Multiple inheritance on public implementation shall not be allowed: <code><public_base_class></code> is not an interface. • Multiple inheritance on protected implementation shall not be allowed : <code><protected_base_class_1></code>. • <code><protected_base_class_2></code> are not interfaces.
88.1	A stateful virtual base shall be explicitly declared in each derived class that accesses it.	
89	A base class shall not be both virtual and nonvirtual in the same hierarchy.	
94	An inherited nonvirtual function shall not be redefined in a derived class.	Does not report for destructor. Message in report file: Inherited nonvirtual function %s shall not be redefined in a derived class.
95	An inherited default parameter shall never be redefined.	
96	Arrays shall not be treated polymorphically.	Reports pointer arithmetic and array like access on expressions whose pointed type is used as a base class.
97	Arrays shall not be used in interface.	Only to prevent array-to-pointer-decay. Not checked on private methods
97.1	Neither operand of an equality operator (== or !=) shall be a pointer to a virtual member function.	Reports == and != on pointer to member function of polymorphic classes (cannot determine statically if it is virtual or not), except when one argument is the null constant.

Namespaces

N.	JSF++ Definition	Polyspace Implementation
98	Every nonlocal name, except <code>main()</code> , should be placed in some namespace.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
99	Namespaces will not be nested more than two levels deep.	

Templates

N.	JSF++ Definition	Polyspace Implementation
104	A template specialization shall be declared before its use.	Reports the actual compilation error message.

Functions

N.	JSF++ Definition	Polyspace Implementation
107	Functions shall always be declared at file scope.	
108	Functions with variable numbers of arguments shall not be used.	
109	A function definition should not be placed in a class specification unless the function is intended to be inlined.	Reports when "inline" is not in the definition of a member function inside the class definition.
110	Functions with more than 7 arguments will not be used.	
111	A function shall not return a pointer or reference to a non-static local object.	Simple cases without alias effect detected.
113	Functions will have a single exit point.	Reports first return, or once per function.
114	All exit points of value-returning functions shall be through return statements.	
116	Small, concrete-type arguments (two or three words in size) should be passed by value if changes made to formal parameters should not be reflected in the calling function.	Report constant parameters references with <code>sizeof <= 2 * sizeof(int)</code> . Does not report for copy-constructor.
119	Functions shall not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).	<p>The checker reports each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported.</p> <p>You can calculate the total number of recursion cycles using the code complexity metric <code>Number of Recursions</code>. Note that unlike the checker, the metric also considers implicit calls, for instance, to compiler-generated constructors during object creation.</p>
121	Only functions with 1 or 2 statements should be considered candidates for inline functions.	Reports inline functions with more than 2 statements.

Comments

N.	JSF++ Definition	Polyspace Implementation
126	Only valid C++ style comments (//) shall be used.	
133	Every source file will be documented with an introductory comment that provides information on the file name, its contents, and any program-required information (e.g. legal statements, copyright information, etc).	<p>Reports when a file does not begin with two comment lines.</p> <p>Note: This rule cannot be annotated in the source code.</p>

Declarations and Definitions

N.	JSF++ Definition	Polyspace Implementation
135	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
136	Declarations should be at the smallest feasible scope.	<p>Reports when:</p> <ul style="list-style-type: none"> • A global variable is used in only one function. • A local variable is not used in a statement (expr, return, init ...) of the same level of its declaration (in the same block) or is not used in two sub-statements of its declaration. <p>Note</p> <ul style="list-style-type: none"> • Non-used variables are reported. • Initializations at definition are ignored (not considered an access)
137	All declarations at file scope should be static where possible.	
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.	Done with Non-initialized variable checks in the software.
144	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	This covers partial initialization.
145	In an enumerator list, the '=' construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	Generates one report for an enumerator list.

Types

N.	JSF++ Definition	Polyspace Implementation
147	The underlying bit representations of floating point numbers shall not be used in any way by the programmer.	Reports on casts with float pointers (except with void*).
148	Enumeration types shall be used instead of integer types (and constants) to select from a limited series of choices.	Reports when non enumeration types are used in switches.

Constants

N.	JSF++ Definition	Polyspace Implementation
149	Octal constants (other than zero) shall not be used.	
150	Hexadecimal constants will be represented using all uppercase letters.	
151	Numeric values in code will not be used; symbolic values will be used instead.	Reports direct numeric constants (except integer/float value 1, 0) in expressions, non - const initializations. and switch cases. char constants are allowed. Does not report on templates non-type parameter. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
151.1	A string literal shall not be modified.	The rule checker flags assignment of string literals to: <ul style="list-style-type: none"> Pointers other than pointers to const objects. Arrays that are not const-qualified.

Variables

N.	JSF++ Definition	Polyspace Implementation
152	Multiple variable declarations shall not be allowed on the same line.	Reports when two consecutive declaration statements are on the same line (unless the statements are part of a macro definition).

Unions and Bit Fields

N.	JSF++ Definition	Polyspace Implementation
153	Unions shall not be used.	
154	Bit-fields shall have explicitly unsigned integral or enumeration types only.	
156	All the members of a structure (or class) shall be named and shall only be accessed via their names.	Reports unnamed bit-fields (unnamed fields are not allowed).

Operators

N.	JSF++ Definition	Polyspace Implementation
157	The right hand operand of a && or operator shall not contain side effects.	Assumes rule 159 is not violated. Messages in report file: <ul style="list-style-type: none"> The right hand operand of a && operator shall not contain side effects. The right hand operand of a operator shall not contain side effects.
158	The operands of a logical && or shall be parenthesized if the operands contain binary operators.	Messages in report file: <ul style="list-style-type: none"> The operands of a logical && shall be parenthesized if the operands contain binary operators. The operands of a logical shall be parenthesized if the operands contain binary operators. Exception for: X Y Z , Z&&Y &&Z
159	Operators , &&, and unary & shall not be overloaded.	Messages in report file: <ul style="list-style-type: none"> Unary operator & shall not be overloaded. Operator shall not be overloaded. Operator && shall not be overloaded.
160	An assignment expression shall be used only as the expression in an expression statement.	Only simple assignment, not +=, ++, etc.
162	Signed and unsigned values shall not be mixed in arithmetic or comparison operations.	
163	Unsigned arithmetic shall not be used.	
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.

N.	JSF++ Definition	Polyspace Implementation
170	More than 2 levels of pointer indirection shall not be used.	Only reports on variables/parameters.
171	Relational operators shall not be applied to pointer types except where both operands are of the same type and point to: <ul style="list-style-type: none"> • the same object, • the same function, • members of the same object, or • elements of the same array (including one past the end of the same array). 	Reports when relational operator are used on pointer types (casts ignored).
173	The address of an object with automatic storage shall not be assigned to an object which persists after the object has ceased to exist.	
174	The null pointer shall not be de-referenced.	Done with checks in software.
175	A pointer shall not be compared to NULL or be assigned NULL; use plain 0 instead.	Reports usage of NULL macro in pointer contexts.
176	A typedef will be used to simplify program syntax when declaring function pointers.	Reports non-typedef function pointers, or pointers to member functions for types of variables, fields, parameters. Returns type of function, cast, and exception specification.

Type Conversions

N.	JSF++ Definition	Polyspace Implementation
177	User-defined conversion functions should be avoided.	Reports user defined conversion function, non-explicit constructor with one parameter or default value for others (even undefined ones). Does not report copy-constructor. Additional message for constructor case: This constructor should be flagged as "explicit".
178	Down casting (casting from base to derived class) shall only be allowed through one of the following mechanism: <ul style="list-style-type: none"> • Virtual functions that act like dynamic casts (most likely useful in relatively simple cases). • Use of the visitor (or similar) pattern (most likely useful in complicated cases). 	Reports explicit down casting, dynamic_cast included. (Visitor patten 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.

N.	JSF++ Definition	Polyspace Implementation
180	Implicit conversions that may result in a loss of information shall not be used.	<p>Reports the following implicit casts :</p> <pre>integer => smaller integer unsigned => smaller or eq signed signed => smaller or eq un-signed integer => float float => integer</pre> <p>Does not report for cast to <code>bool</code> reports for implicit cast on constant done with the option <code>-scalar-overflows-checks signed-and-unsigned</code></p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>
181	Redundant explicit casts will not be used.	Reports useless cast: <code>cast T to T</code> . Casts to equivalent typedefs are also reported.
182	Type casting from any type to or from pointers shall not be used.	Does not report when Rule 181 applies.
184	Floating point numbers shall not be converted to integers unless such a conversion is a specified algorithmic requirement or is necessary for a hardware interface.	Reports <code>float->int</code> conversions. Does not report implicit ones.
185	C++ style casts (<code>const_cast</code> , <code>reinterpret_cast</code> , and <code>static_cast</code>) shall be used instead of the traditional C-style casts.	

Flow Control Standards

N.	JSF++ Definition	Polyspace Implementation
186	There shall be no unreachable code.	<p>Done with gray checks in the software.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>
187	All non-null statements shall potentially have a side-effect.	
188	Labels will not be used, except in switch statements.	
189	The <code>goto</code> statement shall not be used.	
190	The <code>continue</code> statement shall not be used.	
191	The <code>break</code> statement shall not be used (except to terminate the cases of a switch statement).	
192	All <code>if, else if</code> constructs will contain either a final <code>else</code> clause or a comment indicating why a final <code>else</code> clause is not necessary.	<code>else if</code> should contain an <code>else</code> clause.

N.	JSF++ Definition	Polyspace Implementation
193	Every non-empty <code>case</code> clause in a <code>switch</code> statement shall be terminated with a <code>break</code> statement.	
194	All <code>switch</code> statements that do not intend to test for every enumeration value shall contain a final <code>default</code> clause.	Reports only for missing <code>default</code> .
195	A <code>switch</code> expression will not represent a Boolean value.	
196	Every <code>switch</code> statement will have at least two cases and a potential <code>default</code> .	
197	Floating point variables shall not be used as loop counters.	Assumes 1 loop parameter.
198	The initialization expression in a <code>for</code> loop will perform no actions other than to initialize the value of a single <code>for</code> loop parameter.	Reports if <code>loop</code> parameter cannot be determined. Assumes Rule 200 is not violated. The <code>loop variable</code> parameter is assumed to be a variable.
199	The increment expression in a <code>for</code> loop will perform no action other than to change a single loop parameter to the next value for the loop.	Assumes 1 loop parameter (Rule 198), with non class type. Rule 200 must not be violated for this rule to be reported.
200	Null initialize or increment expressions in <code>for</code> loops will not be used; a <code>while</code> loop will be used instead.	
201	Numeric variables being used within a <code>for</code> loop for iteration counting shall not be modified in the body of the loop.	Assumes 1 loop parameter (AV rule 198), and no alias writes.

Expressions

N.	JSF++ Definition	Polyspace Implementation
202	Floating point variables shall not be tested for exact equality or inequality.	Reports only direct equality/inequality. Check done for all expressions.
203	Evaluation of expressions shall not lead to overflow/underflow.	Done with overflow checks in the software.
204	A single operation with side-effects shall only be used in the following contexts: <ul style="list-style-type: none"> • by itself • the right-hand side of an assignment • a condition • the only argument expression with a side-effect in a function call • condition of a loop • switch condition • single part of a chained operation 	Reports when: <ul style="list-style-type: none"> • A side effect is found in a return statement • A side effect exists on a single value, and only one operand of the function call has a side effect.

N.	JSF++ Definition	Polyspace Implementation
204.1	The value of an expression shall be the same under any order of evaluation that the standard permits.	Reports when: <ul style="list-style-type: none"> • Variable is written more than once in an expression • Variable is read and write in sub-expressions • Volatile variable is accessed more than once Note Read-write operations such as ++, are only considered as a write.
205	The volatile keyword shall not be used unless directly interfacing with hardware.	Reports if volatile keyword is used.

Memory Allocation

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

Fault Handling

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

Portable Code

N.	JSF++ Definition	Polyspace Implementation
209	The basic types of 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.
215	Pointer arithmetic will not be used.	Reports: p + Ip - Ip++p--p+=p-= Allows p[i].

Unsupported JSF++ Rules

- "Code Size and Complexity" on page 13-60

- “Rules” on page 13-60
- “Environment” on page 13-61
- “Libraries” on page 13-61
- “Header Files” on page 13-61
- “Style” on page 13-61
- “Classes” on page 13-61
- “Namespaces” on page 13-63
- “Templates” on page 13-63
- “Functions” on page 13-63
- “Comments” on page 13-64
- “Initialization” on page 13-64
- “Types” on page 13-64
- “Unions and Bit Fields” on page 13-64
- “Operators” on page 13-64
- “Type Conversions” on page 13-64
- “Expressions” on page 13-65
- “Memory Allocation” on page 13-65
- “Portable Code” on page 13-65
- “Efficiency Considerations” on page 13-65
- “Miscellaneous” on page 13-65
- “Testing” on page 13-66

Code Size and Complexity

N.	JSF++ Definition
2	There shall not be any self-modifying code.

Rules

N.	JSF++ Definition
4	To break a “should” rule, the following approval must be received by the developer: <ul style="list-style-type: none"> • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool)
5	To break a “will” or a “shall” rule, the following approvals must be received by the developer: <ul style="list-style-type: none"> • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool) • approval from the software product manager (obtained by the unit approval in the developmental CM tool)
6	Each deviation from a “shall” rule shall be documented in the file that contains the deviation. Deviations from this rule shall not be allowed, AV Rule 5 notwithstanding.

N.	JSF++ Definition
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.

N.	JSF++ Definition
69	A member function that does not affect the state of an object (its instance variables) will be declared const. Member functions should be const by default. Only when there is a clear, explicit reason should the const modifier on member functions be omitted.
70	A class will have friends only when a function or object requires access to the private elements of the class, but is unable to be a member of the class for logical or efficiency reasons.
70.1	An object shall not be improperly used before its lifetime begins or after its lifetime ends.
71	Calls to an externally visible operation of an object, other than its constructors, shall not be allowed until the object has been fully initialized.
72	<p>The invariant for a class should be:</p> <ul style="list-style-type: none"> • A part of the postcondition of every class constructor, • A part of the precondition of the class destructor (if any), • A part of the precondition and postcondition of every other publicly accessible operation.
73	Unnecessary default constructors shall not be defined.
77	A copy constructor shall copy all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).
80	The default copy and assignment operators will be used for classes when those operators offer reasonable semantics.
84	Operator overloading will be used sparingly and in a conventional manner.
85	When two operators are opposites (such as == and !=), both will be defined and one will be defined in terms of the other.
86	Concrete types should be used to represent simple independent concepts.
87	Hierarchies should be based on abstract classes.
90	Heavily used interfaces should be minimal, general and abstract.
91	Public inheritance will be used to implement “is-a” relationships.
92	<p>A subtype (publicly derived classes) will conform to the following guidelines with respect to all classes involved in the polymorphic assignment of different subclass instances to the same variable or parameter during the execution of the system:</p> <ul style="list-style-type: none"> • Preconditions of derived methods must be at least as weak as the preconditions of the methods they override. • Postconditions of derived methods must be at least as strong as the postconditions of the methods they override. <p>In other words, subclass methods must expect less and deliver more than the base class methods they override. This rule implies that subtypes will conform to the Liskov Substitution Principle.</p>
93	“has-a” or “is-implemented-in-terms-of” relationships will be modeled through membership or non-public inheritance.

Namespaces

N.	JSF++ Definition
100	<p>Elements from a namespace should be selected as follows:</p> <ul style="list-style-type: none"> • using declaration or explicit qualification for few (approximately five) names, • using directive for many names.

Templates

N.	JSF++ Definition
101	<p>Templates shall be reviewed as follows:</p> <ol style="list-style-type: none"> 1 with respect to the template in isolation considering assumptions or requirements placed on its arguments. 2 with respect to all functions instantiated by actual arguments.
102	Template tests shall be created to cover all actual template instantiations.
103	Constraint checks should be applied to template arguments.
105	A template definition's dependence on its instantiation contexts should be minimized.
106	Specializations for pointer types should be made where appropriate.

Functions

N.	JSF++ Definition
112	Function return values should not obscure resource ownership.
115	If a function returns error information, then that error information will be tested.
117	<p>Arguments should be passed by reference if NULL values are not possible:</p> <ul style="list-style-type: none"> • 117.1 - An object should be passed as <code>const T&</code> if the function should not change the value of the object. • 117.2 - An object should be passed as <code>T&</code> if the function may change the value of the object.
118	<p>Arguments should be passed via pointers if NULL values are possible:</p> <ul style="list-style-type: none"> • 118.1 - An object should be passed as <code>const T*</code> if its value should not be modified. • 118.2 - An object should be passed as <code>T*</code> if its value may be modified.
120	Overloaded operations or methods should form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal parameters.
122	Trivial accessor and mutator functions should be inlined.
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
127	Code that is not used (commented out) shall be deleted. Note: This rule cannot be annotated in the source code.
128	Comments that document actions or sources (e.g. tables, figures, paragraphs, etc.) outside of the file being documented will not be allowed.
129	Comments in header files should describe the externally visible behavior of the functions or classes being documented.
130	The purpose of every line of executable code should be explained by a comment, although one comment may describe more than one line of code.
131	One should avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).
132	Each variable declaration, typedef, enumeration value, and structure member will be commented.
134	Assumptions (limitations) made by functions should be documented in the function's preamble.

Initialization

N.	JSF++ Definition
143	Variables will not be introduced until they can be initialized with meaningful values. (See also AV Rule 136, AV Rule 142, and AV Rule 73 concerning declaration scope, initialization before use, and default constructors respectively.)

Types

N.	JSF++ Definition
146	Floating point implementations shall comply with a defined floating point standard. The standard that will be used is the ANSI/IEEE® Std 754 [1].

Unions and Bit Fields

N.	JSF++ Definition
155	Bit-fields will not be used to pack data into a word for the sole purpose of saving space.

Operators

N.	JSF++ Definition
167	The implementation of integer division in the chosen compiler shall be determined, documented and taken into account.

Type Conversions

N.	JSF++ Definition
183	Every possible measure should be taken to avoid type casting.

Expressions

N.	JSF++ Definition
204	<p>A single operation with side-effects shall only be used in the following contexts:</p> <ol style="list-style-type: none"> 1 by itself 2 the right-hand side of an assignment 3 a condition 4 the only argument expression with a side-effect in a function call 5 condition of a loop 6 switch condition 7 single part of a chained operation

Memory Allocation

N.	JSF++ Definition
207	Unencapsulated global data will be avoided.

Portable Code

N.	JSF++ Definition
210	Algorithms shall not make assumptions concerning how data is represented in memory (e.g. big endian vs. little endian, base class subobject ordering in derived classes, nonstatic data member ordering across access specifiers, etc.).
210.1	Algorithms shall not make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier.
211	Algorithms shall not assume that shorts, ints, longs, floats, doubles or long doubles begin at particular addresses.
212	Underflow or overflow functioning shall not be depended on in any special way.
214	Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.

Efficiency Considerations

N.	JSF++ Definition
216	Programmers should not attempt to prematurely optimize code.

Miscellaneous

N.	JSF++ Definition
217	Compile-time and link-time errors should be preferred over run-time errors.
218	Compiler warning levels will be set in compliance with project policies.

Testing

N.	JSF++ Definition
219	All tests applied to a base class interface shall be applied to all derived class interfaces as well. If the derived class poses stronger postconditions/invariants, then the new postconditions /invariants shall be substituted in the derived class tests.
220	Structural coverage algorithms shall be applied against flattened classes.
221	Structural coverage of a class within an inheritance hierarchy containing virtual functions shall include testing every possible resolution for each set of identical polymorphic references.

Configure Verification of Modules or Libraries

- “Provide Context for C Code Verification” on page 14-2
- “Provide Context for C++ Code Verification” on page 14-4
- “Verify C Application Without main Function” on page 14-6
- “Verify C++ Classes” on page 14-9

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 <code>main</code> must call first.
Functions to call (-main-generator-calls)	Specify the functions that the generated <code>main</code> 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 14-6

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 <code>main</code> must call.
Functions to call within the specified classes (-class-analyzer-calls)	Specify methods that the generated <code>main</code> must call.
Analyze class contents only (-class-only)	Specify that the generated <code>main</code> must call class methods only.
Skip member initialization check (-no-constructors-init-check)	Specify that the generated <code>main</code> 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 14-9

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 <code>main</code> .
Verify module or library (-main-generator)	<p>Before verification, Polyspace checks if your code contains a <code>main</code> function.</p> <p>If a <code>main</code> function exists, the software uses that <code>main</code>. Otherwise, the software generates a <code>main</code> using the options that you specify:</p> <ul style="list-style-type: none"> • 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 14-2

Verify C++ Classes

In this section...
“Verification of Classes” on page 14-9
“Methods and Class Specifics” on page 14-11

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	<p>Unless explicitly initialized, in each method, global variables can have any value allowed by their type.</p> <p>For instance, in the following code, Polyspace assumes that <code>globvar1</code> can have any value allowed by its type. Therefore, an orange Division by zero appears on the division by <code>globvar1</code>. However, because <code>globvar2</code> is explicitly initialized, the Division by zero check on division by <code>globvar2</code> is green.</p> <pre>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); } }; };</pre>

Code Construct	Assumption
Classes with undefined constructors	<p>The members of the classes can be non-initialized.</p> <p>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.</p> <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>

Methods and Class Specifics

- “Simple Class” on page 14-11
- “Template Classes” on page 14-13
- “Abstract Classes” on page 14-13
- “Static Classes” on page 14-14
- “Inherited Classes” on page 14-14
- “Simple Inheritance” on page 14-15
- “Multiple Inheritance” on page 14-16
- “Virtual Inheritance” on page 14-17
- “Class Integration” on page 14-17

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)`.

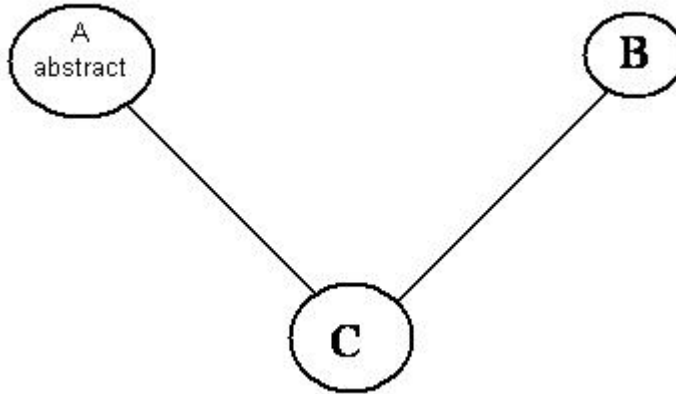
Abstract Classes

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;
  
```



```

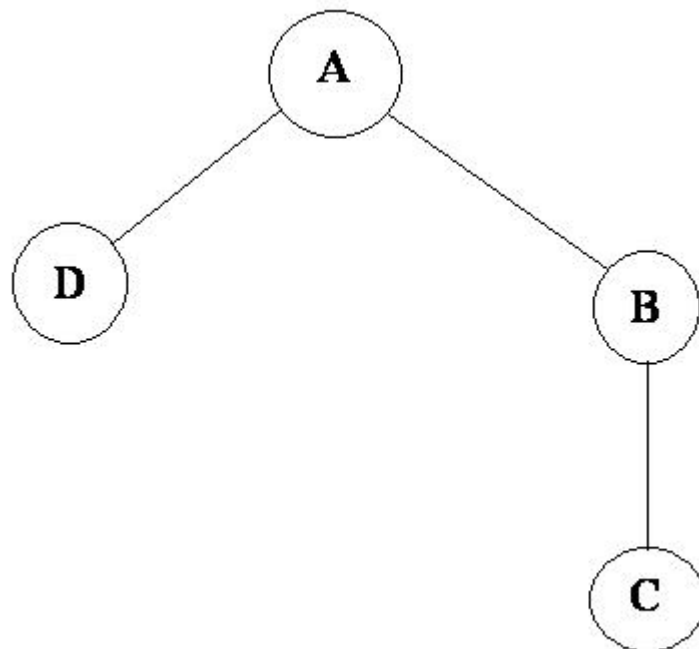
public :
    Point(int intx, int inty) : Location (intx, inty)
    {
        visible = false;
    };
    void show(void) { visible = true;};
    void hide(void) { visible = false;};
    bool isvisible(void) {return visible;};
};

```

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.

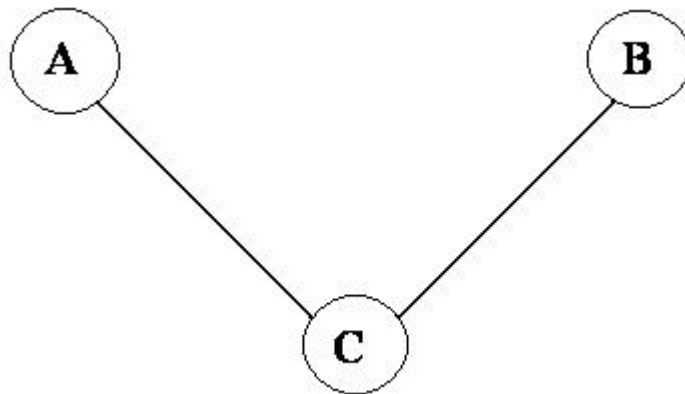
In a case such a this, Polyspace software allows you to run the following verifications:

- 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



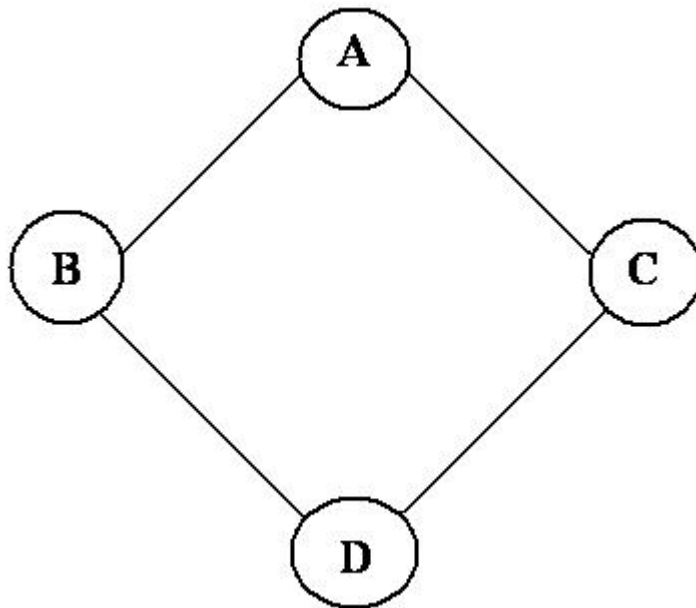
Consider the following classes:

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 14-4

Configure Comment Import from Previous Results

- “Import Review Information from Previous Polyspace Analysis” on page 15-2
- “Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results” on page 15-5

Import Review Information from Previous Polyspace Analysis

After you have reviewed analysis results, you can reuse information from the review for subsequent analyses. If you specify a result status or severity or add notes in your results file, they carry over to the results of the next analysis on the same project. If you add the same information as comments to your code (annotate), they carry over to any subsequent analysis of the code, whether in the same project or not. You can also hide results using code annotations. For more information on commenting, see “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

This topic shows how to import review information from one result file to another. Importing the review information saves you from reviewing already justified results. For instance, after you import the information, on the **Results List** pane (user interface of desktop products), clicking the  icon skips justified results. Using this icon, you can browse through unreviewed results. You can also filter the justified checks from display.

Automatic Import from Last Analysis

By default, in the Polyspace user interface (desktop products only), review information is imported automatically from the most recent analysis on the project module. You can disable this default behavior.

- 1 Select **Tools > Preferences**, which opens the Polyspace Preferences dialog box.
- 2 Select the **Project and Results Folder** tab.
- 3 Under **Import Comments**, clear **Automatically import comments from last verification**.
- 4 Click **OK**.

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

If you run analysis at the command line (and do not upload results to the Polyspace Access web interface), you have to explicitly import from another set of results. See “Command Line” on page 15-3.

Import from Another Analysis Result

You can import review information directly from another Polyspace result to the current result.

If a result is found in both a Bug Finder and Code Prover analysis, you can add review information to the Bug Finder result and import to the Code Prover result. For instance, most coding rule checkers are common to Bug Finder and Code Prover. You can add review information to coding rule violations in Bug Finder and import to the same violations in Code Prover.

User Interface (Desktop Products Only)

To import review information from another set of results:

- 1 Open the current analysis results.
- 2 Select **Tools > Import Comments**.
- 3 Navigate to the folder containing your previous results.
- 4 Select the other results file (with extension `.psbf` or `.pscp`) and then click **Open**.

The review information from the previous results are imported into the current results.

Command Line

Use the option `-import-comments` during analysis to import comments from a previous verification.

To import review information from multiple results, use the `polyspace-comments-import` command.

Import Algorithm

You can directly import review information from another set of results into the current results. However, it is possible that part of your review information is not imported to a subsequent analysis because:

- You have changed your source code so that the line with a previous result is not exactly identical to the line in the current run.

The comment import tool accounts for additional code that simply shifts an existing line. For instance, the tool recognizes that line 10 in Run 1 and line 12 in Run 2 have the same statement. If a division by zero occurs on line 10 in Run 1 and you have not fixed the issue in Run 2, the result along with associated review information are imported to line 12 in Run 2.

- Run 1:

```
10 baseLine = min/numRecipients;
11
12
```

- Run 2:

```
10 /* Calculate a baseline per recipient
11    based on minimum available resources */
12 baseLine = min/numRecipients;
```

However, if you change the line content itself, for instance, change `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.

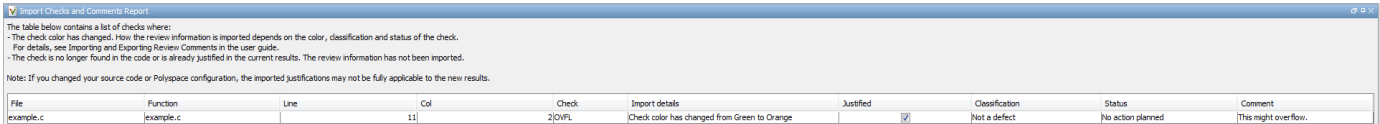
View Imported Review Information That Does Not Apply

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

Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results

When you check your code for MISRA C: 2012 violations, Polyspace imports justifications of MISRA C: 2004 violations from previous analyses (if they exist). You can upgrade from checking of MISRA C: 2004 rules to MISRA C: 2012 rules while retaining your justifications. For general rules on comment import, see “Import Review Information from Previous Polyspace Analysis” on page 15-2.

The software maps MISRA C: 2004 **Status**, **Severity**, and **Comment** values that you added through the user interface or code annotations to the corresponding MISRA C: 2012 results, if the results exist. For more information about mapping, consult addendum one of the MISRA C: 2012 publication.

Type	Check: (9)	Status	Severity	Comment: (9)
MISRA C: 2004	6.3 Typedefs that indicate size and sig...	Unreviewed	Unset	MISRA2004-6.3 comment
MISRA C: 2004	6.3 Typedefs that indicate size and sig...	To fix	Medium	MISRA2004-6.3
MISRA C: 2004	8.1 Functions shall have prototype de...	To fix	Low	MISRA2004-8.1
MISRA C: 2004	11.3 A cast should not be performed b...	Justified	Low	MISRA2004-11.3
MISRA C: 2004	11.4 A cast should not be performed b...	Unreviewed	Unset	MISRA2004-11.4 comment
MISRA C: 2004	12.12 The underlying bit representatio...	Unreviewed	Unset	MISRA2004-12.12 comm...
MISRA C: 2004	13.2 Tests of a value against zero sho...	Not a defect	Low	MISRA2004-13.2
MISRA C: 2004	14.4 The goto statement shall not be ...	Not a defect	Low	MISRA2004-14.4
MISRA C: 2004	14.9 An if (expression) construct shall ...	Not a defect	Low	MISRA2004-13.2
MISRA C: 2004	19.5 Macros shall not be #define'd an...	Justified	Low	MISRA2004-19.5

If you are transitioning from MISRA C: 2004 to MISRA C: 2012, you do not have to review results that you have already justified.

Type	Check	Status	Severity	Comment: (7)
MISRA C: 2012	Dir 4.6 typedefs that indicate size and...	Unreviewed	Unset	MISRA2004-6.3 comment
MISRA C: 2012	Dir 4.6 typedefs that indicate size and...	To fix	Medium	MISRA2004-6.3
MISRA C: 2012	8.4 A compatible declaration shall be v...	To fix	Low	MISRA2004-8.1
MISRA C: 2012	11.3 A cast shall not be performed bet...	Unreviewed	Unset	MISRA2004-11.4 comment
MISRA C: 2012	11.4 A conversion should not be perfo...	Justified	Low	MISRA2004-11.3
MISRA C: 2012	14.4 The controlling expression of an i...	Not a defect	Low	MISRA2004-13.2
MISRA C: 2012	15.1 The goto statement should not b...	Not a defect	Low	MISRA2004-14.4
MISRA C: 2012	15.6 The body of an iteration-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 18-6


Interpret Polyspace Code Prover Results

- “Interpret Polyspace Code Prover Results” on page 16-2
- “Code Prover Result and Source Code Colors” on page 16-8
- “Code Prover Run-Time Checks” on page 16-13
- “Dashboard” on page 16-16
- “Concurrency Modeling” on page 16-20
- “Results List” on page 16-21
- “Source” on page 16-24
- “Result Details” on page 16-30
- “Call Hierarchy” on page 16-33
- “Variable Access” on page 16-36
- “Code Prover Analysis Following Red and Orange Checks” on page 16-42
- “Order of Code Prover Run-Time Checks” on page 16-46
- “Orange Checks in Code Prover” on page 16-48
- “Managing Orange Checks” on page 16-51
- “Critical Orange Checks” on page 16-55
- “Limit Display of Orange Checks” on page 16-57
- “Software Quality Objectives” on page 16-60
- “Reduce Orange Checks” on page 16-67
- “Test Orange Checks for Run-Time Errors” on page 16-70
- “Limitations of Automatic Orange Tester” on page 16-73

Interpret Polyspace Code Prover Results

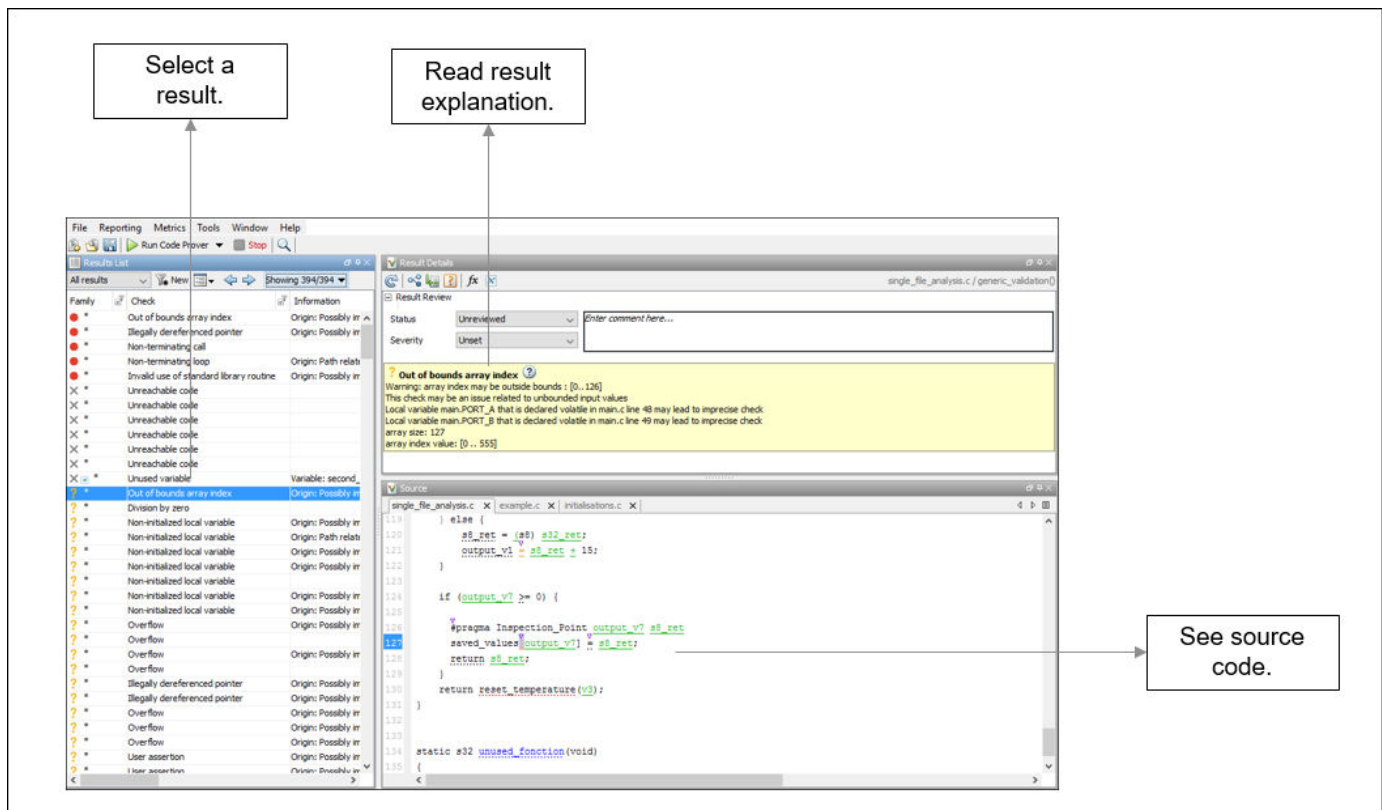
When you open the results of a Polyspace Code Prover analysis, you see a list on the **Results List** pane. The list consists of run-time checks, coding rule violations, code metrics and global variable usage.

You can first narrow down the focus of your review:

- Use filters on the results list columns. For instance, you can 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” on page 19-2. Once you narrow down the list, you can begin reviewing individual results. This topic describes how to review a result.



The screenshot displays the Polyspace Code Prover interface. On the left, the **Results List** pane shows a list of error messages. The first item, **Out of bounds array index**, is selected. Above this pane, a box labeled **Select a result.** has an arrow pointing to the selected item. On the right, the **Result Review** pane shows the details of the selected error. It includes a warning: **Out of bounds array index**. Below the warning, there is a source code snippet. A box labeled **See source code.** has an arrow pointing to the source code snippet. Above the Result Review pane, a box labeled **Read result explanation.** has an arrow pointing to the warning text.

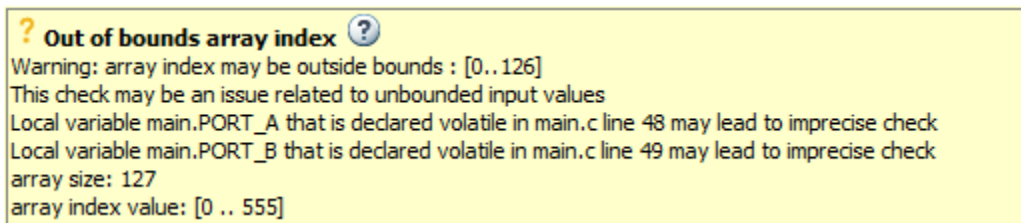
To begin your review, select a result in the list.

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.



The message consists of several parts:

- Check color and icon: See “Code Prover Result and Source Code Colors” on page 16-8. 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.

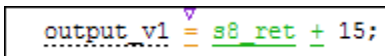
In the example, the message states the array size (127), the array bounds (0..126), and the range of values that the array index variable can take at that point in the code (0..555).

- 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.




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.

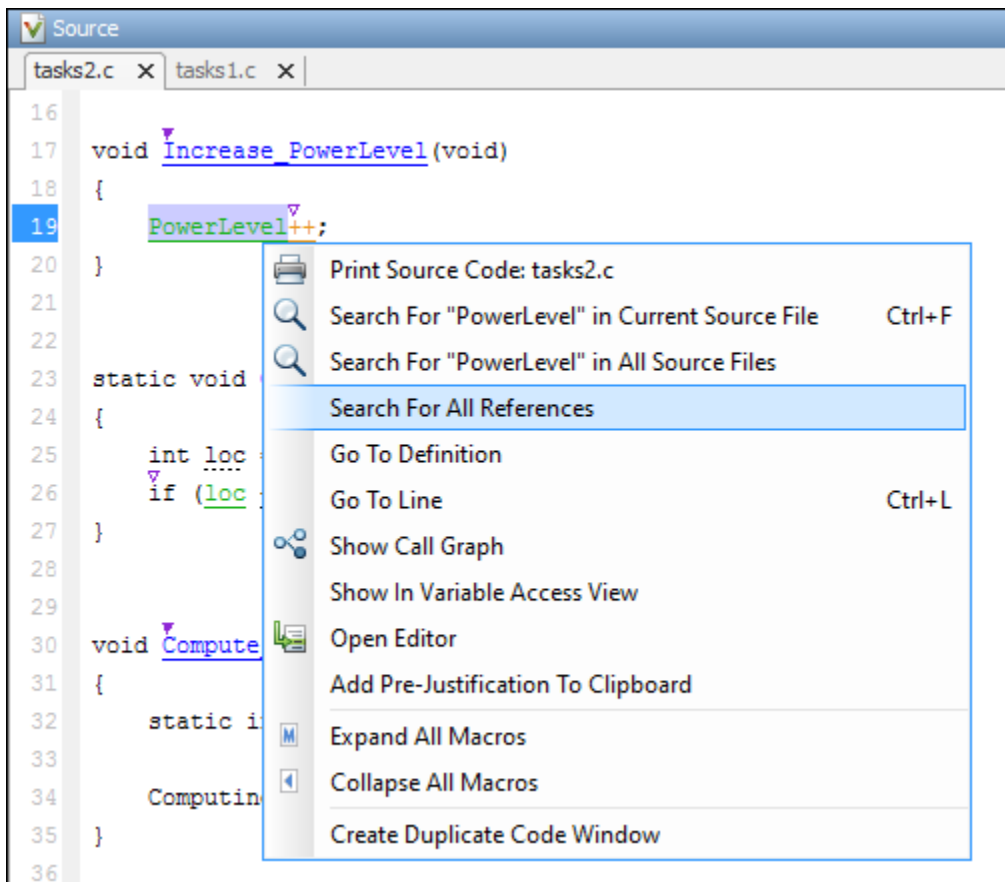
Alternatively, click the  icon to see a graphical representation of the call sequence leading to the result. To navigate to functions in this sequence, click through nodes in the graph.

- 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.

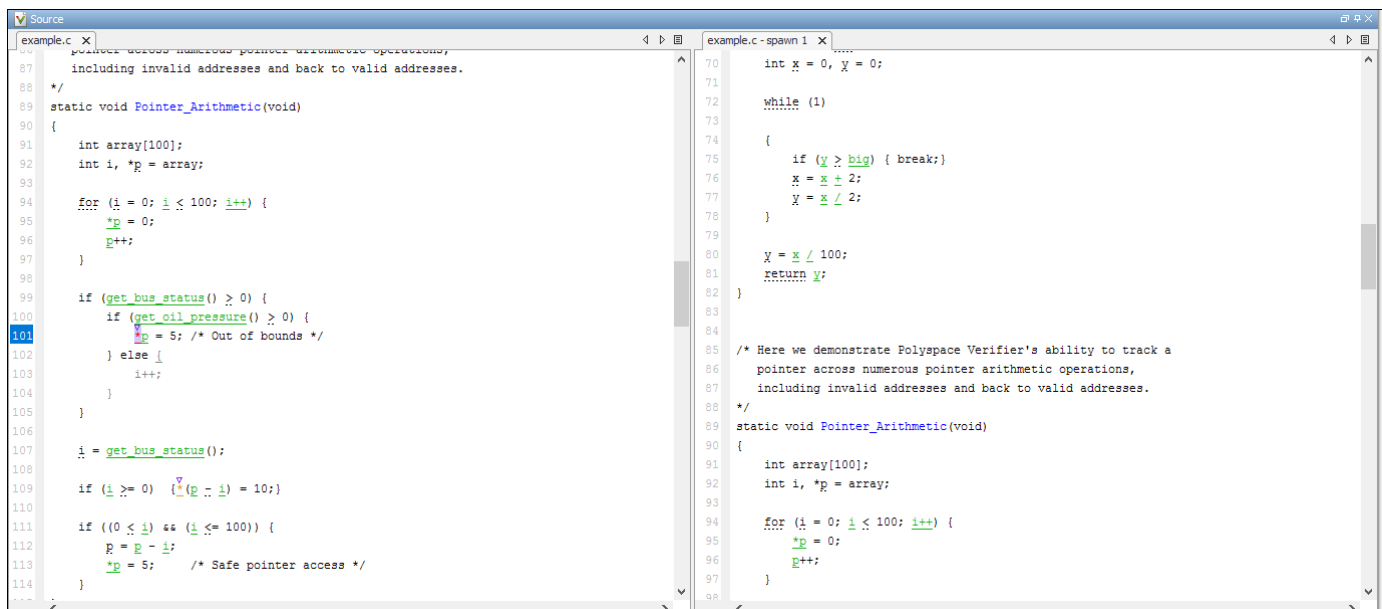
you might want to navigate from a function call to the function definition.

For other examples of what to look for, see “Code Prover Run-Time Checks” on page 16-13. 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” on page 19-2
- “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2

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.

Family:...	Check
	Division by zero
 *	Unreachable code
	Unreachable code
	Out of bounds array index
	Overflow
	Overflow
	Overflow
	Overflow
	Overflow
	Non-initialized local variable
	Non-initialized local variable
	Overflow
	Overflow

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	<p>Highlights operations that are proven to cause a particular error on all execution paths*.</p> <p>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".</p>	<p>Red Overflow on:</p> <pre>z = x+y;</pre> <p>The operation + overflows for every value of x and y that the verification considers at that point.</p>	

Check Color	Purpose	Example	Icon
Gray	Highlights unreachable code.	Gray Unreachable code check: <pre>if(x>0) {} else {} </pre> The <code>else</code> branch is unreachable for all values of <code>x</code> 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 Code Prover” on page 16-48.	Orange Overflow on: <pre>z = x+y;</pre> The analysis could not prove whether the operation <code>+</code> overflows. The most common reason is that the operation overflows only for some values of <code>x</code> and <code>y</code> that the verification considers at that point. You can use the tooltips on the variables <code>x</code> and <code>y</code> 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: <pre>z = x+y;</pre> The operation <code>+</code> does not overflow for all values of <code>x</code> and <code>y</code> 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 16-42.

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 ✕☒ 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:*

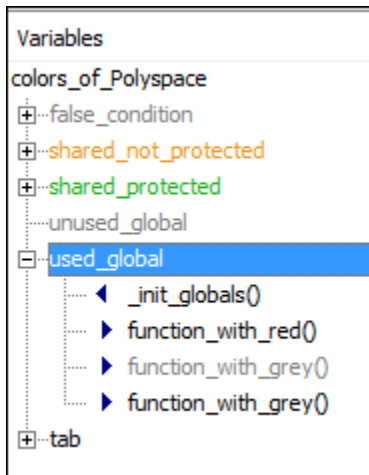
Polyspace assigns a lighter shade of gray to code deactivated due to conditional compilation. Such code occurs, for instance, in `#ifdef` statements where the macro for a branch is not defined. This code does not affect the verification.

```
#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” on page 16-36.

Code Prover Run-Time Checks

Polyspace Code Prover checks each operation in your 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 16-8.

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

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 17-11
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 17-13
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 17-35
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 17-38
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 17-40
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 17-63
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 17-68

Numerical Checks

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 17-7
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 17-26

Check	How to Review	Details
Overflow	Review prior operations in your code that lead to an operation overflowing.	“Review and Fix Overflow Checks” on page 17-57

Static Memory Checks

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 17-2
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 17-16
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 17-53

Control Flow Checks

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 17-42
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 17-46

C++ Checks

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 17-24
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 17-15
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 17-22

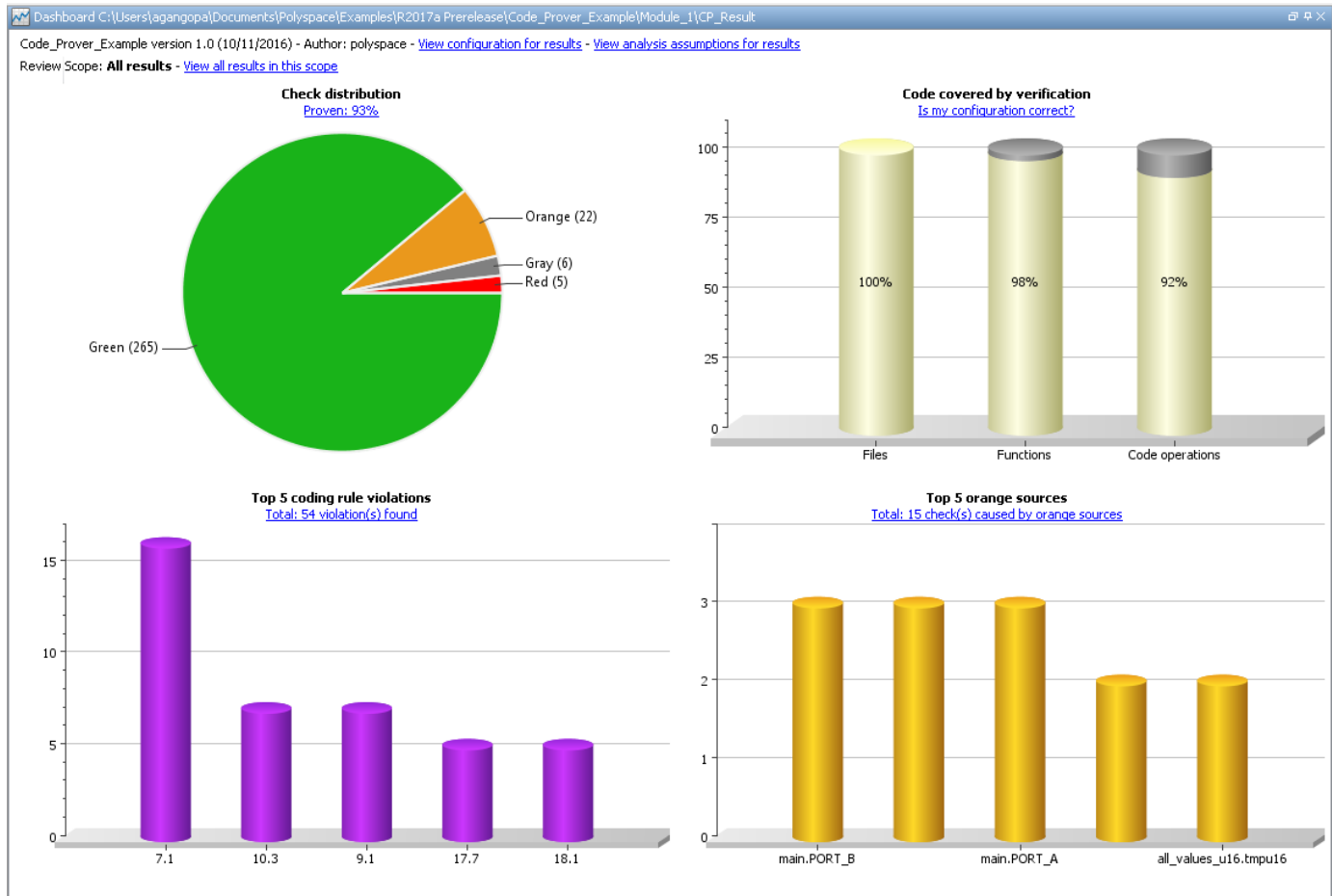
Check	How to Review	Details
Null this-pointer calling method	Investigate why the pointer to the current object can be NULL-valued.	“Review and Fix Null This-pointer Calling Method Checks” on page 17-51
Uncaught exception	Investigate how an exception can escape uncaught from the function where it is thrown.	“Review and Fix Uncaught Exception Checks” on page 17-66

Other Checks

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 17-3
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 17-30
User assertion	Investigate why the condition in an <code>assert</code> statement fails.	“Review and Fix User Assertion Checks” on page 17-72

Dashboard

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.

Unreachable functions		
Unreachable function (19/44)	File	Line
unused_fonction()	single_file_analysis.c	134
orderregulate()	tasks1.c	35
Tserver()	tasks1.c	73
tregulate()	tasks1.c	61
initregulate()	tasks1.c	47
proc1()	tasks1.c	101
proc2()	tasks1.c	107
server1()	tasks1.c	95
server2()	tasks1.c	89
End_CS()	tasks2.c	79
Computing_from_Sensors()	tasks2.c	23
Pilot_Balance()	tasks2.c	54
Exec_One_Cycle()	tasks2.c	66
Increase_PowerLevel()	tasks2.c	17
Begin_CS()	tasks2.c	74
Get_PowerLevel()	tasks2.c	38
Sequencer()	tasks2.c	60
Command_Ordering()	tasks2.c	46
Compute_Injection()	tasks2.c	30

[? Reasons for Unchecked Code](#)

Close

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 22-72.

- **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 16-8.

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” on page 16-67.

- **Top 5 orange sources**

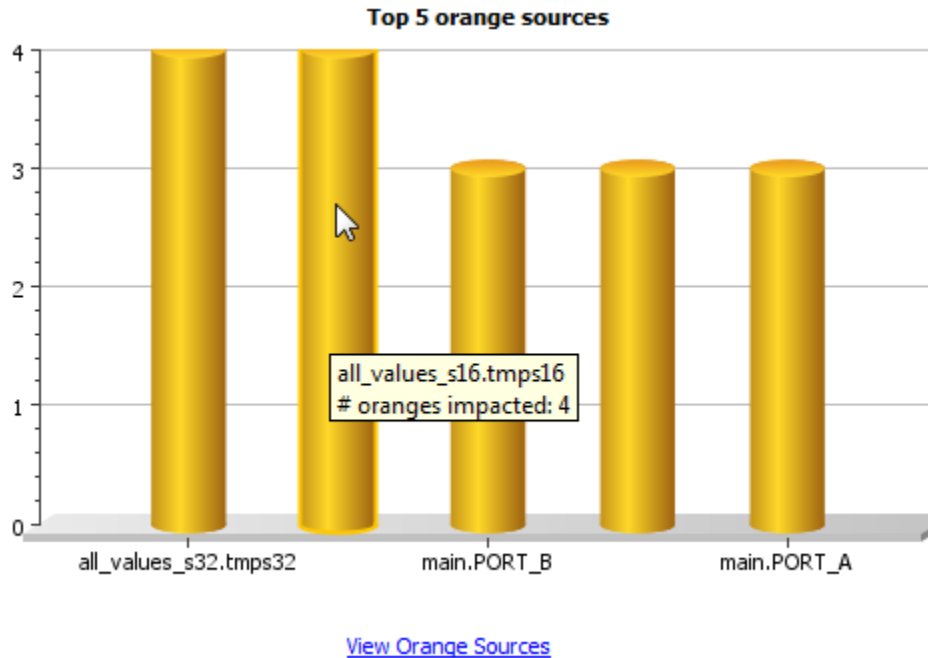
An orange source is a variable or function that leads to an orange check. This column graph displays five orange sources affecting the most number of checks.

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 10-4.
- **Top 5 coding rule violations**

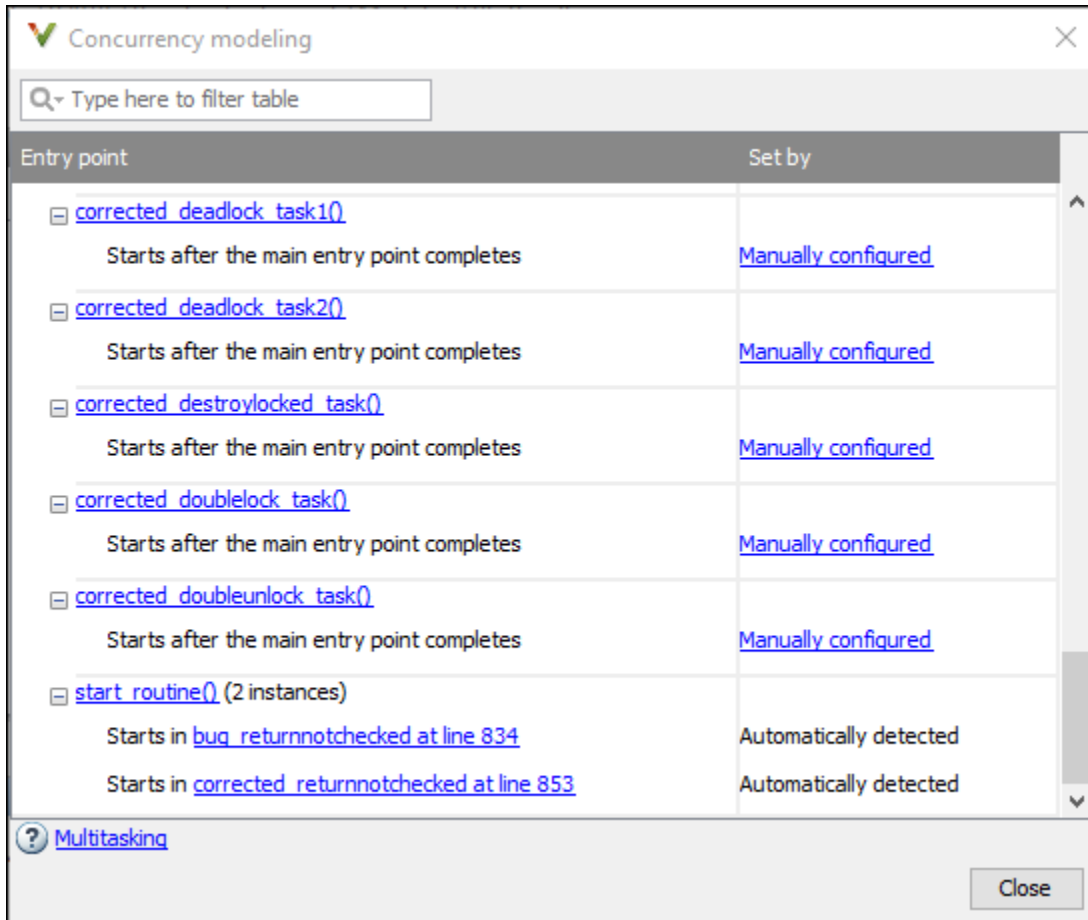
This column graph displays the five most violated coding rules. Each column represents a coding rule and is indexed by the rule number. The height of the column indicates the number of violations of the coding rule represented by that column.

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” on page 19-2.
- 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 16-20**.

Concurrency Modeling

The **Concurrency Modeling** view displays all the tasks and interrupts that the analysis extracts from your code and your Polyspace multitasking configuration.



in the table, the functions are listed in the first column by order of decreasing priority. The second column shows how Polyspace detects each task or interrupt: automatically, manually from the Polyspace configuration, or from an external file.

From this view, you can:


- Click a function name to go to its definition in the source code.
- Click an event to go to the corresponding call to the concurrency primitive in the source code, for instance `pthread_create`.
- Click **Manually configured**, for functions that are manually configured, to go to the **Multitasking** node on the **Configuration** pane.


Results List

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	<p>Category of the result, for instance:</p> <ul style="list-style-type: none"> For run-time checks: Groups such as static memory, numerical, control flow, etc. For coding rule violations: Groups defined by the coding rule standard. <p>For instance, MISRA C: 2012 defines groups related to code constructs such as functions, pointers and arrays, etc.</p>
Check	<p>Result name, for instance:</p> <ul style="list-style-type: none"> For run-time checks: Check name For coding rule violations: Coding rule number
Detail	<p>Additional information about a result. The column shows the first line of the Result Details pane.</p> <p>For an example of how to use this column, see the result MISRA C:2012 Dir 1.1.</p>
Information	<p>For orange checks, this column indicates whether the check is related to path or input values. For more information, see “Critical Orange Checks” on page 16-55.</p> <p>For coding rule violations, this column indicates whether the rule belongs to the Required subset.</p> <p>For global variables, this column contains the global variable name.</p>
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 <i>class_name::function_name</i> .
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: <ul style="list-style-type: none"> • Unset • High • Medium • Low
Status	Review status you have assigned to the result. The possible statuses are: <ul style="list-style-type: none"> • Unreviewed (default status) • To investigate • To fix • Justified • No action planned • Not a defect • Other

Attribute	Description
Justified	<p>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.</p> <p>If you choose the option File from the  list, this column indicates the percentage of checks that you have justified per file and function.</p>
Comments	Comments you have entered about the result
Assigned to	<p>User name of reviewer assigned to this result.</p> <p>This column is visible only for results that you open from Polyspace Access.</p>
Ticket Key	<p>When you create a bug tracking tool (BTT) ticket for a result, this field contains the ticket ID. Click the ticket ID in the Results Details to open the ticket in the BTT interface.</p> <p>This column is visible only for results that you open from Polyspace Access.</p>

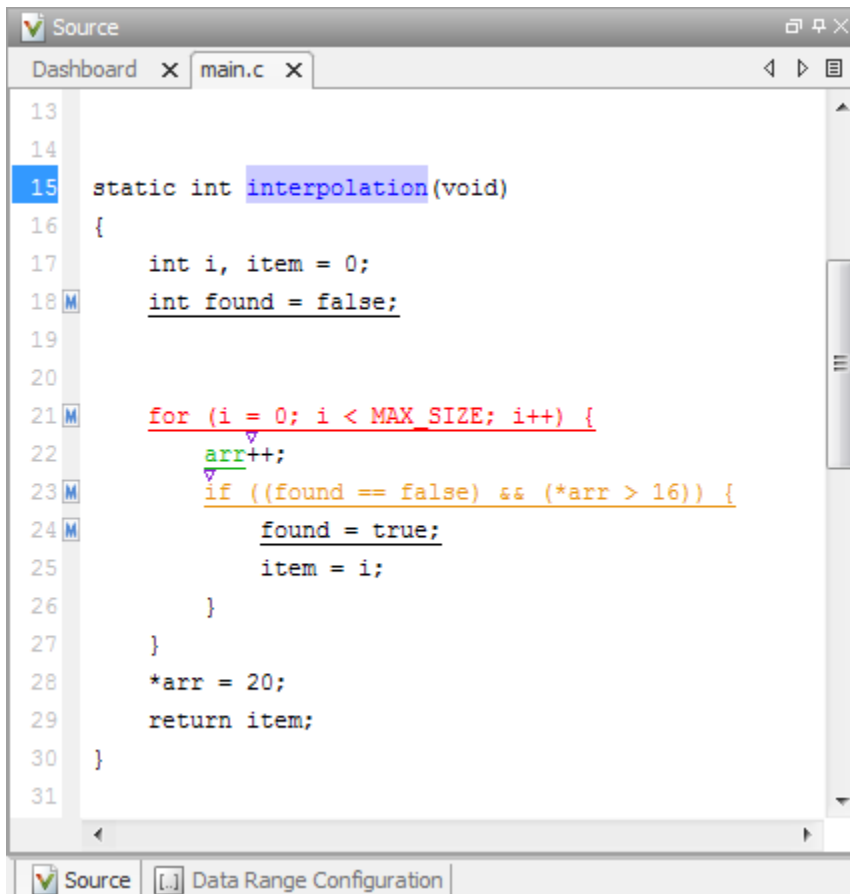
To show or hide any of the columns, right-click anywhere on the column titles. From the context menu, select or clear the title of the column that you want to show or hide.

Using this pane, you can:

- Navigate through the results.
- Organize your result review using filters on the columns. For more information, see “Filter and Group Results” (Polyspace Bug Finder).

Source

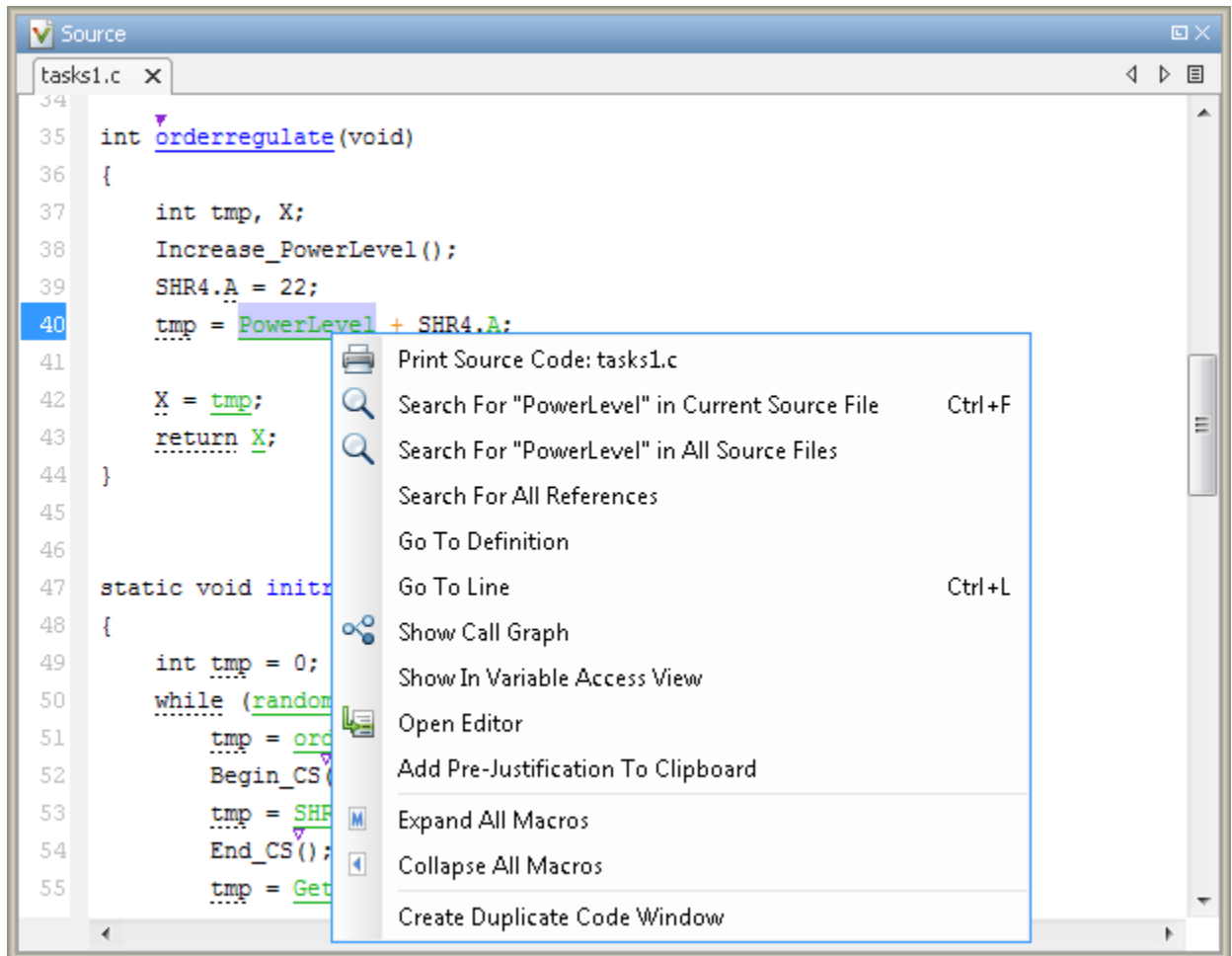
The **Source** pane shows the source code with the results highlighted with specific colors and icons. For more information, see “Code Prover Result and Source Code Colors” on page 16-8.



On the **Source** pane, you can:

- **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 Range**

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.

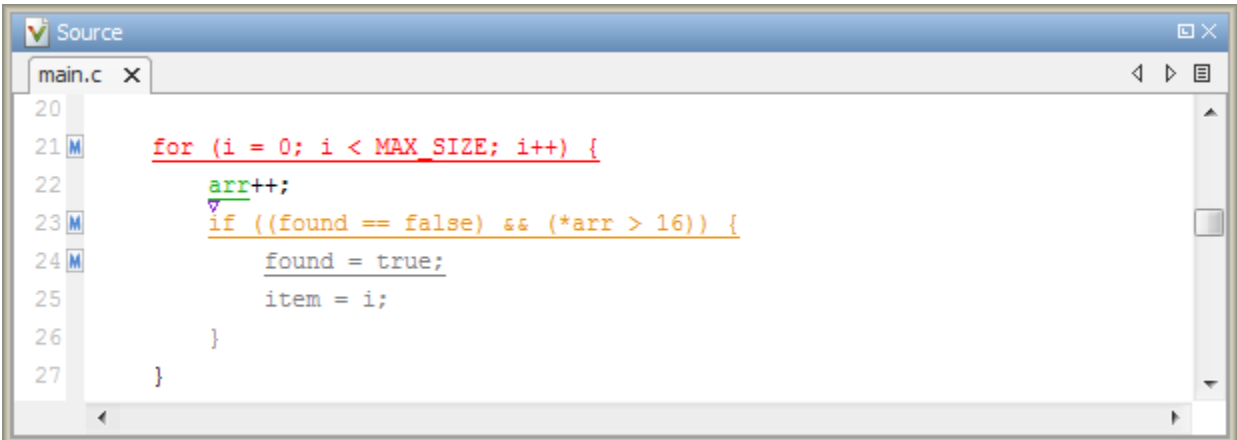
However, the displayed range can also include additional values because of approximation.

- 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.

- **Expand 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.

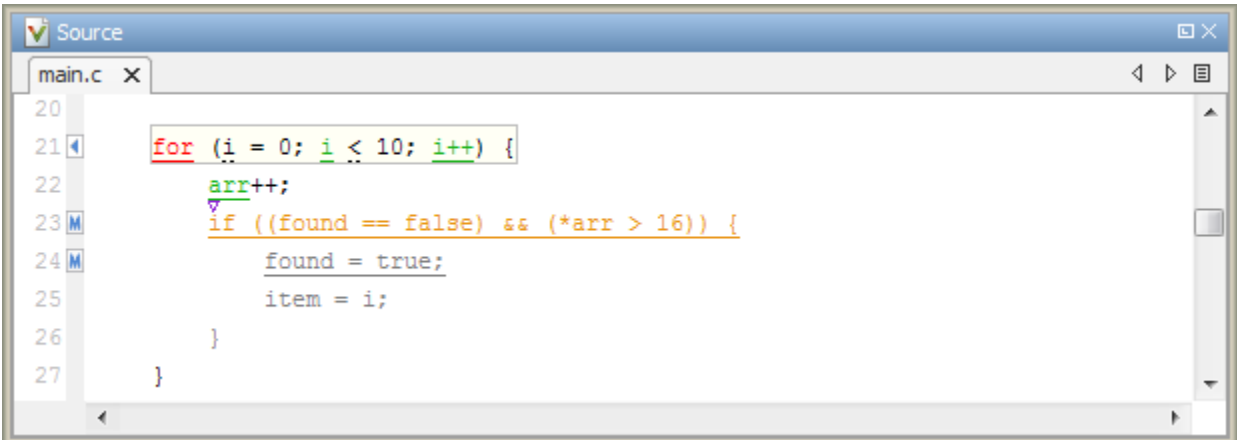


```

20
21 M   for (i = 0; i < MAX_SIZE; i++) {
22     arr++;
23 M   if ((found == false) && (*arr > 16)) {
24     found = true;
25     item = i;
26   }
27   }

```

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



```

20
21 M   for (i = 0; i < 10; i++) {
22     arr++;
23 M   if ((found == false) && (*arr > 16)) {
24     found = true;
25     item = i;
26   }
27   }

```

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**.

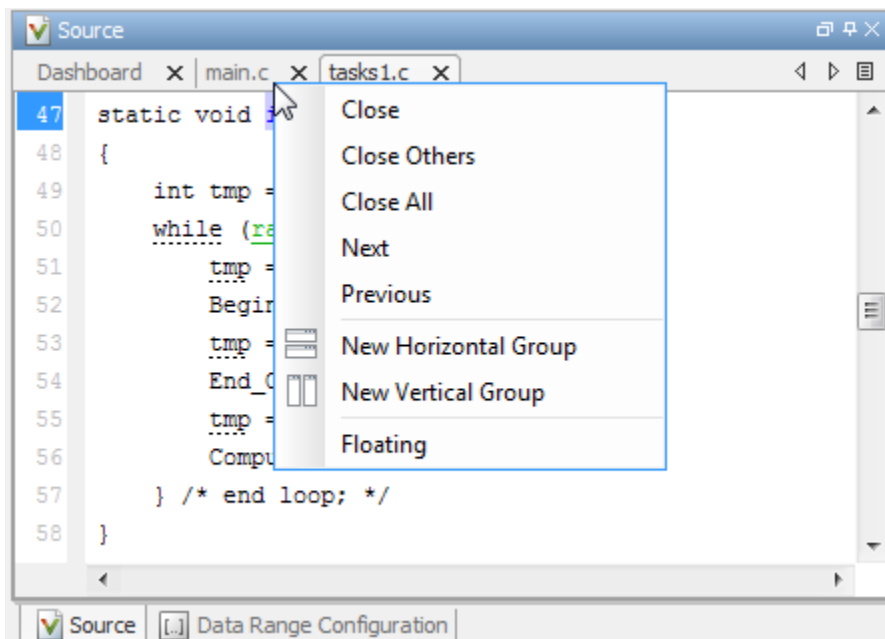
Note

- 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.
-

• 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 \times 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.
- **View Code Block**

On the **Source** pane, to highlight a block of code, click either its opening or closing brace.

```

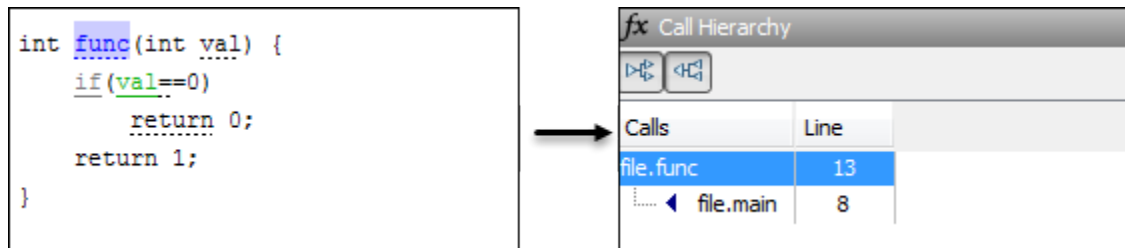
47 static void initregulate(void)
48 {
49     int tmp = 0;
50     while (random_int() < 1000) {
51         tmp = orderregulate();
52         Begin_CS();
53         tmp = SHR + SHR2 + SHR6;
54         End_CS();
55         tmp = Get_PowerLevel();
56         Compute_Injection();
57     } /* end loop; */
58 }

```

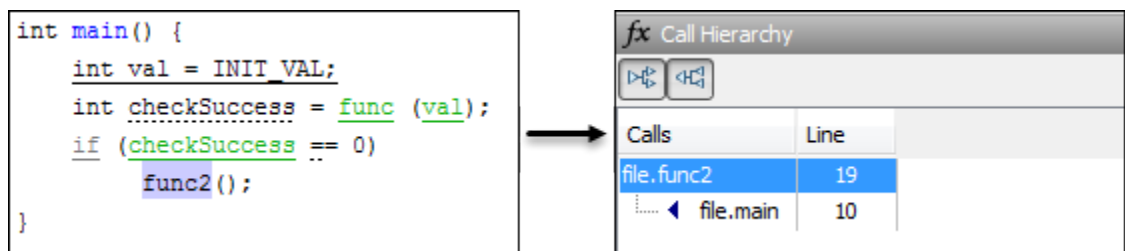
- **See Function Callers and Callees**

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.



- 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.



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

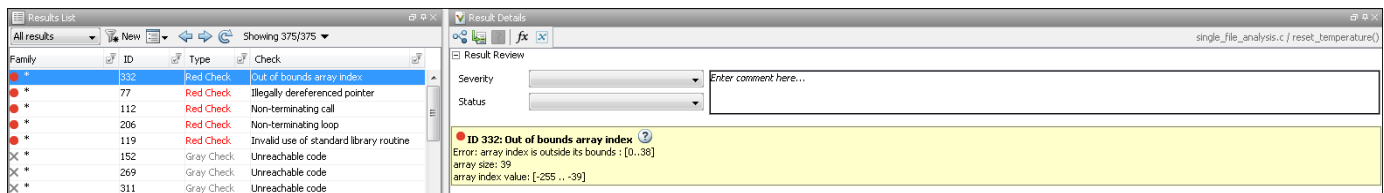
On the **Results List** pane, if you select a check, you see additional information on the **Result Details** pane.

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:

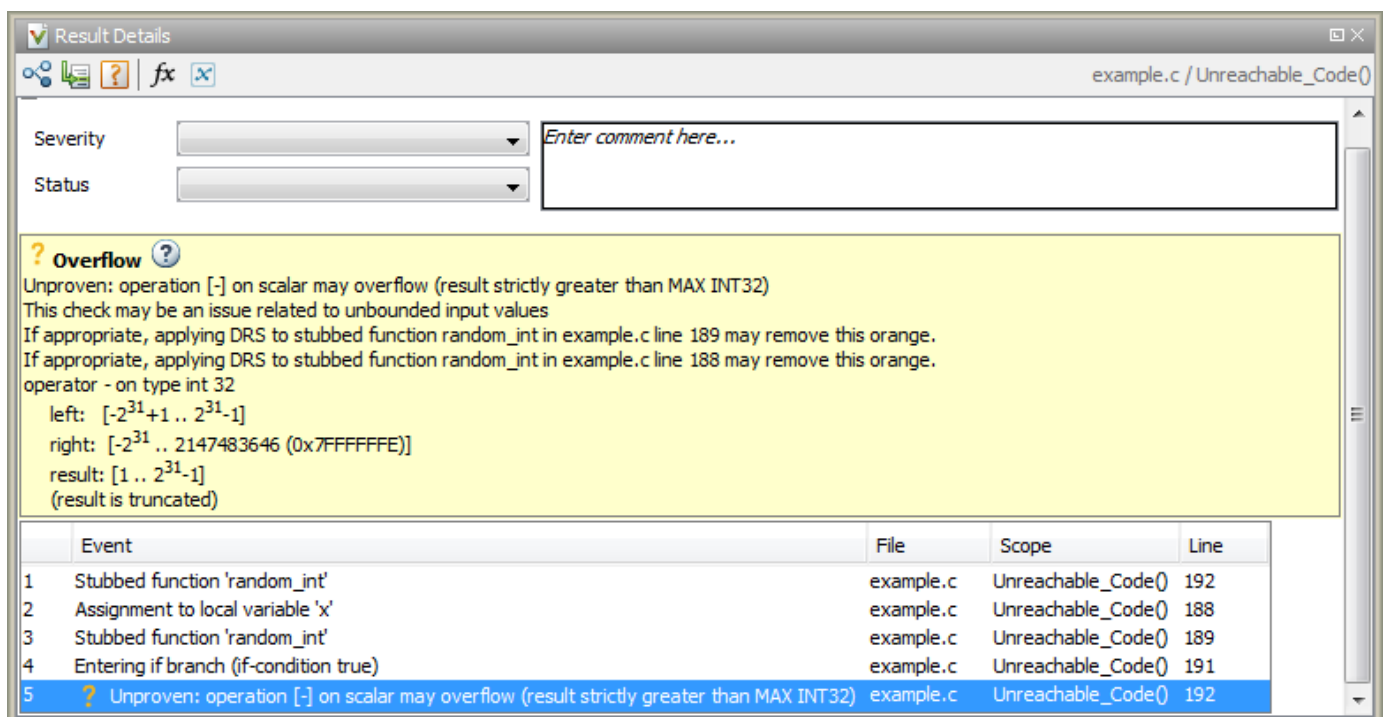
- Assign a reviewer to the result. A reviewer can filter the **Results List** to only show results that are assigned to him or her.
- Create a ticket in a bug tracking tool (BTT) such as JIRA. Once you create the ticket the **Results Details** for this defect shows the ticket ID. Click the ID to open the ticket in the BTT interface.

See “Open or Export Results from Polyspace Access” (Polyspace Code Prover Access).



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.

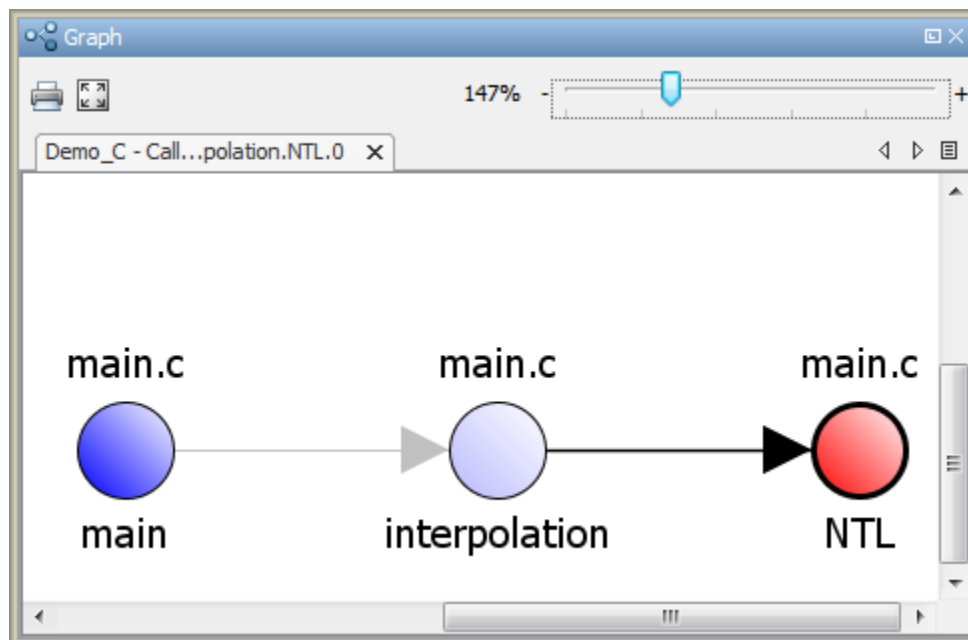


The following columns appear in the traceback:

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.

Show 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.

Show Call Hierarchy and Variable Access

From the **Result Details** pane, you can open the **Call Hierarchy** and **Variable Access** panes.

- Select the  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” on page 16-33.


- 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” on page 16-36.

Call Hierarchy

The **Call Hierarchy** pane displays the call tree of functions in the source code.

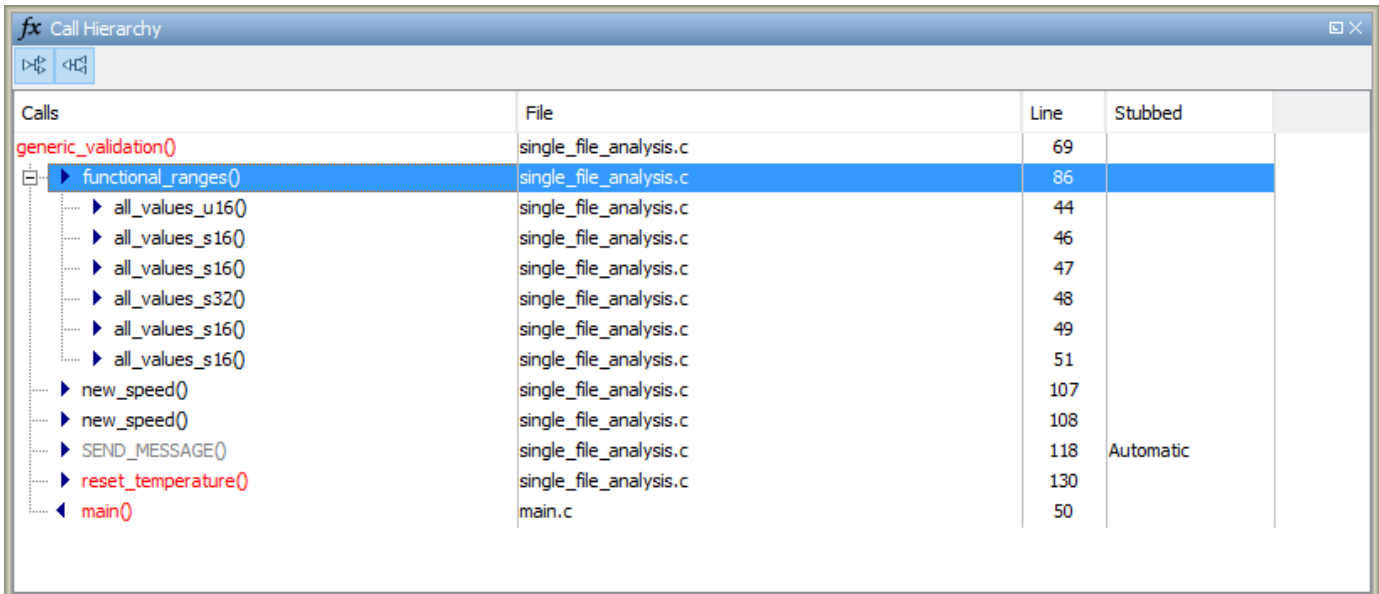
For each function `foo`, the **Call Hierarchy** pane lists the functions and tasks that call `foo` (callers) and those called by `foo` (callees). The callers are indicated by  (functions) or  (tasks). The callees are indicated by  (functions) or  (tasks). The **Call Hierarchy** pane lists direct function calls and indirect calls through function pointers. The indirect calls are shown with the  icon. Calls that are unreachable are shown with the function name in grey.

To open this pane, in the Polyspacedesktop 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.



Calls	File	Line	Stubbed
<code>generic_validation()</code>	single_file_analysis.c	69	
▶ <code>functional_ranges()</code>	single_file_analysis.c	86	
▶ <code>all_values_u16()</code>	single_file_analysis.c	44	
▶ <code>all_values_s16()</code>	single_file_analysis.c	46	
▶ <code>all_values_s16()</code>	single_file_analysis.c	47	
▶ <code>all_values_s32()</code>	single_file_analysis.c	48	
▶ <code>all_values_s16()</code>	single_file_analysis.c	49	
▶ <code>all_values_s16()</code>	single_file_analysis.c	51	
▶ <code>new_speed()</code>	single_file_analysis.c	107	
▶ <code>new_speed()</code>	single_file_analysis.c	108	
▶ <code>SEND_MESSAGE()</code>	single_file_analysis.c	118	Automatic
▶ <code>reset_temperature()</code>	single_file_analysis.c	130	
◀ <code>main()</code>	main.c	50	

The line number in the **Call Hierarchy** pane refers to a different line in the source code:

- For the function name, the line number refers to the beginning of the function definition. The definition of `generic_validation` begins on line 69.
- For a callee name, the number refers to the line where the callee is called. The callee `functional_ranges` is called by `generic_validation` on line 86.

- For a caller name, the number refers to the line where the caller calls the function. The caller `main` calls `generic_validation` on line 50.

Tip To navigate to the call location in the source code, select a caller or callee name

In the **Call Hierarchy** pane, you can perform these actions:

- **Show/Hide Callers and Callees**

Customize the view to display callers only or callees only. Show or hide callers and callees by clicking this button:



- **Navigate Call Hierarchy**

You can navigate the call hierarchy in your source code. For a function, double-click a caller or callee name to navigate to the caller or callee definition in the source code.

- **Determine if Function is Stubbed**

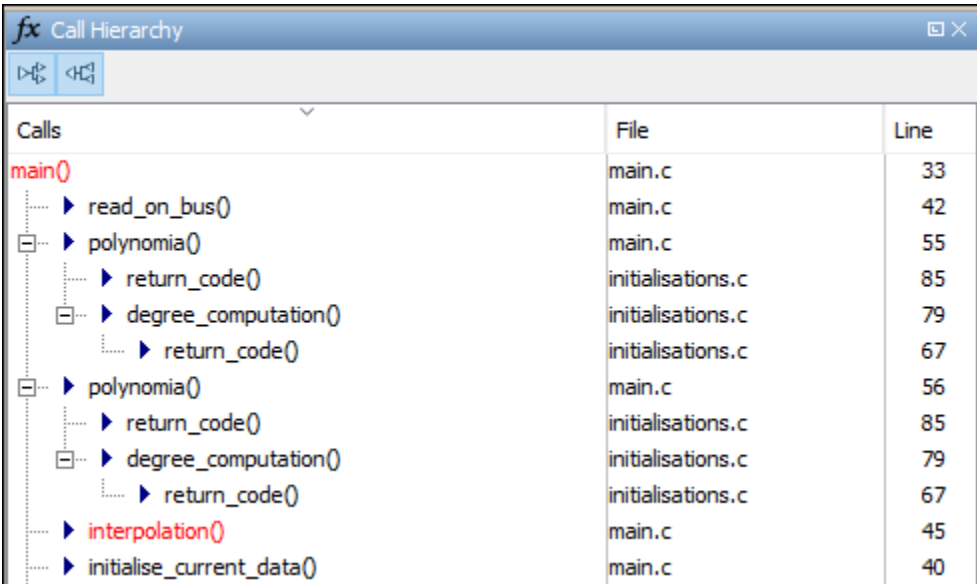
You can determine from the **Stubbed** column if a function is stubbed. The entries in the column show why a function was stubbed.

- **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 “Stubbed Functions”.

- **See 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**.




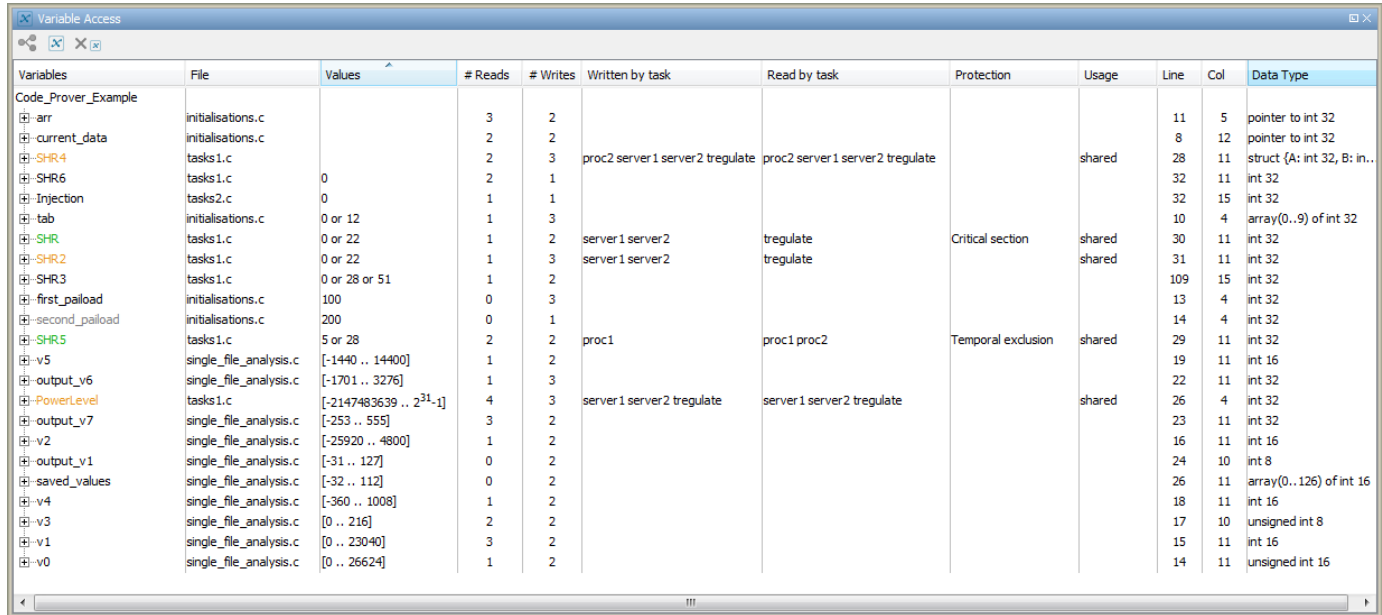
Calls	File	Line
main()	main.c	33
▶ read_on_bus()	main.c	42
▶ polynomial()	main.c	55
▶ return_code()	initialisations.c	85
▶ degree_computation()	initialisations.c	79
▶ return_code()	initialisations.c	67
▶ polynomial()	main.c	56
▶ return_code()	initialisations.c	85
▶ degree_computation()	initialisations.c	79
▶ return_code()	initialisations.c	67
▶ interpolation()	main.c	45
▶ initialise_current_data()	main.c	40

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

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 Polyspacedesktop user interface, select the  button on the **Result Details** pane.



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_payload	initialisations.c	100	0	3					13	4	int 32
second_payload	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: <ul style="list-style-type: none"> • 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	<i>Only for tasks:</i> Name of task performing write access on variable
Read by task	<i>Only for tasks:</i> 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 <code>__polyspace__stdstubs.c</code> , it indicates that the variable is accessed inside a Standard Library function.

For example, consider the global variable, SHR2:

Variables	File	Values	# Reads	# Writes	Written by task	Read by task	Protection	Usage	Line	Col	Data Type
SHR2	tasks1.c	0 or 22	1	3	server1 server2	tregulate		shared	31	11	int 32
server1()	tasks1.c				server1						
server2()	tasks1.c				server2						
tregulate()	tasks1.c					tregulate					
_init_globals()	tasks1.c	0							31	11	
Tserver()	tasks1.c	0							85	4	
initregulate()	tasks1.c	0 or 22							53	20	
Tserver()	tasks1.c	22							76	4	

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.

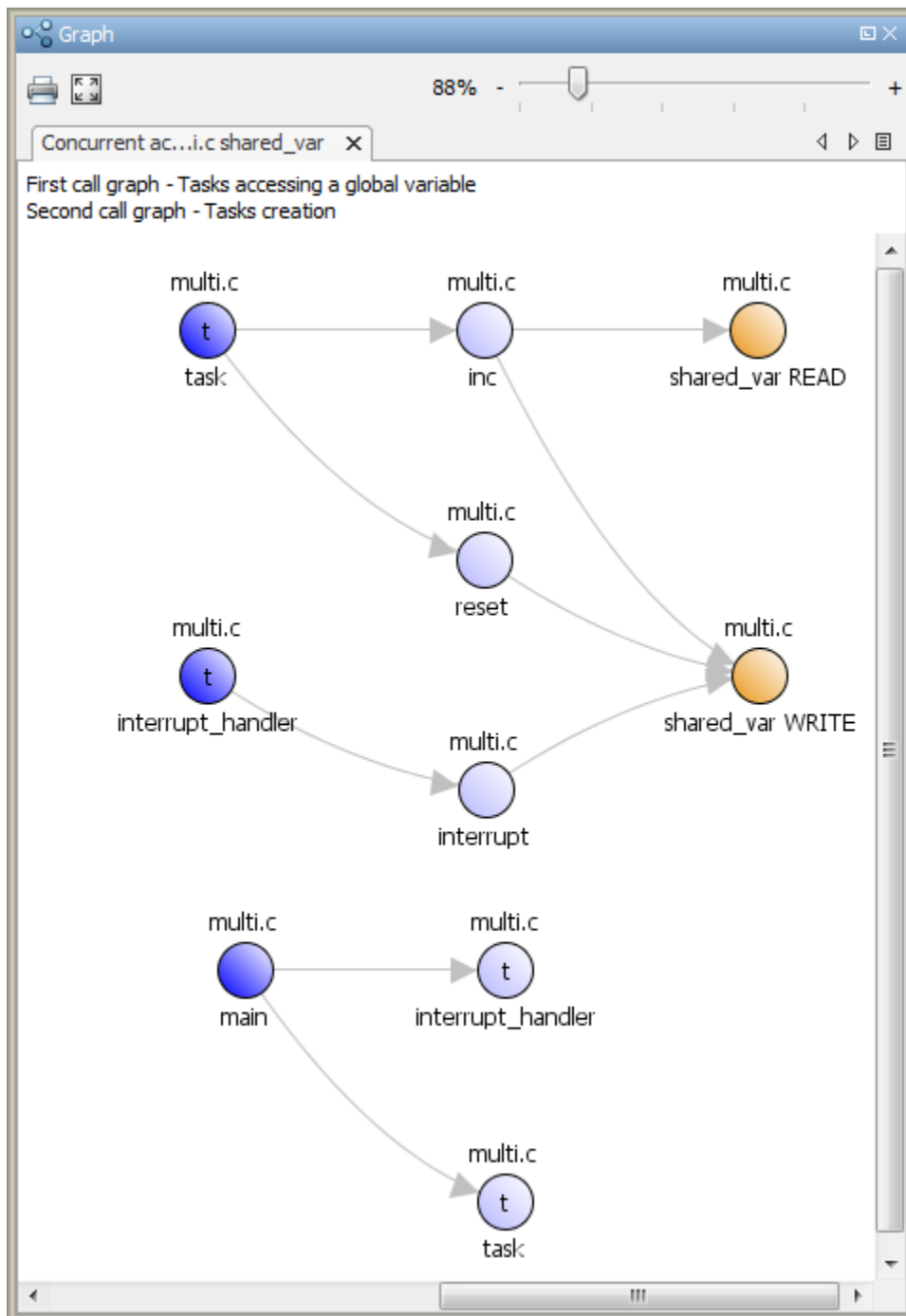
The information about global variables and read/write access operations obtained from the **Variable Access** pane is called the data dictionary.

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.

Variables	File	Values	# Reads	# Writes	Written by task	Read by task	Protection	Usage	Line	Col	Data Type
SHR4	tasks.1.c		2	3	proc2 server1 server2 tregulate	proc2 server1 server2 tregulate		shared	28	11	struct {A: int 32, B: in
proc2()	tasks.1.c				proc2						
server1()	tasks.1.c				server1						
server2()	tasks.1.c				server2						
tregulate()	tasks.1.c				tregulate						
proc2()	tasks.1.c					proc2					
server1()	tasks.1.c					server1					
server2()	tasks.1.c					server2					
tregulate()	tasks.1.c					tregulate					
_init_globals()	tasks.1.c								28	11	
SHR4.B	tasks.1.c		1	1					28	11	int 32
proc2()	tasks.1.c				proc2						
proc2()	tasks.1.c					proc2					
proc2()	tasks.1.c	22							111	9	
proc2()	tasks.1.c	22							112	27	
SHR4.A	tasks.1.c		1	1	server1 server2 tregulate	server1 server2 tregulate		shared	28	11	int 32
server1()	tasks.1.c				server1						
server2()	tasks.1.c				server2						
tregulate()	tasks.1.c				tregulate						
server1()	tasks.1.c					server1					
server2()	tasks.1.c					server2					
tregulate()	tasks.1.c					tregulate					
orderregulate()	tasks.1.c	22							39	9	
orderregulate()	tasks.1.c	22							40	28	

• **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**.

Variables	Values	# Re...	# Wri...	Wri...	Re...	Protection	Usage	Line	Col	File	Data Type
Example.pstanonymous_loc_0x1234		0	0				neither rea...	5	32	Example.cpp	union {


• **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 \ddagger (read) or $\ddot{\ddagger}$ (write). Access through local 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.

Variables	File	Values	# Reads	# Writes	Written by task	Read by task	Protection	Usage
tab	initialisations.c	0 or 12	1	3				
interpolation()	main.c							
_init_globals()	initialisations.c	0						
interpolation()	main.c	0 or 12						


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/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.

- **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 **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`.

Code Prover Analysis Following Red and Orange Checks

Polyspace 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.

Exceptions to this behavior can occur. For instance:

- For an orange 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 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.

Use these two rules to understand your checks. The following examples show how the two rules 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 %%                                %% file2.c %%

void f(int);                                #include <math.h>
int read_an_input(void);

int main() {                                  void f(int a) {
    int x,old_x;                               int tmp;
    x = read_an_input();                       tmp = sqrt(0-a);
    old_x = x;                                  }
    if (x<0 || x>10)
        return 1;
    f(x);
    x = 1 / old_x;
    // Red Division by Zero
    return 0;
}
```

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 `[-231, 231-1]`.
- 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 Checks in Unreachable Code

Code Prover can sometimes show red checks in code that is supposed to be unreachable and gray. When propagating variable ranges, Code Prover sometimes makes approximations. In making these

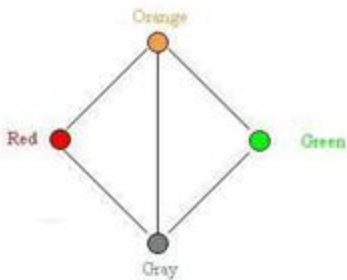
approximations, Code Prover might consider an otherwise unreachable branch as reachable. If an error appears in that unreachable branch, it is colored red.

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.



See Also

Related Examples

- “Interpret Polyspace Code Prover Results” on page 16-2
- “Order of Code Prover Run-Time Checks” on page 16-46

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 16-42.

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`.

The order of checks can be nontrivial if one of the operands in a binary operation is a floating-point variable. Code Prover checks 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.

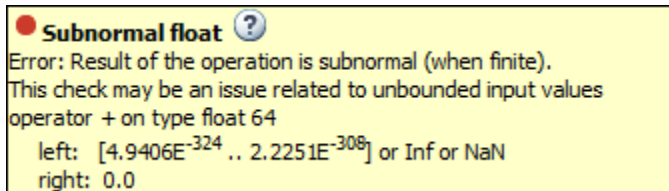
For the next checks, this path is not considered.

- 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, $x > 0$. $\&\& x < \text{DBL_MIN}$. 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`) | Invalid operation on floats | NaNs (`-check-nan`) | Overflow | Subnormal float

More About

- “Code Prover Analysis Following Red and Orange Checks” on page 16-42

Orange Checks in Code Prover

In this section...
“When Orange Checks Occur” on page 16-48
“Why Review Orange Checks” on page 16-49
“How to Review Orange Checks” on page 16-49
“How to Reduce Orange Checks” on page 16-49

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: <ul style="list-style-type: none"> • 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 <code>volatile</code> 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 $\{-1\} \cup [0, 10]$ 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 Polyspace Code Prover Results” on page 16-2.
- 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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

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 16-67.

- 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 16-68.

- 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 16-68.

See Also

More About

- “Managing Orange Checks” on page 16-51
- “Critical Orange Checks” on page 16-55
- “Reduce Orange Checks” on page 16-67
- “Limit Display of Orange Checks” on page 16-57
- “Test Orange Checks for Run-Time Errors” on page 16-70

Managing Orange Checks

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.
- Review all red checks and all orange checks.

To see only red and critical orange checks, from the drop-down list in the left of the **Results List** pane toolbar, select **Critical checks**.

Software Development Stage

Development Stage	Situation	Review Process
Initial stage or unit development stage	<p>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:</p> <ul style="list-style-type: none"> You have not defined all your functions and class methods. You do not have a <code>main</code> function <p>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.</p>	<p>In the initial stages of development, review all red checks. For orange checks, depending on your requirements, do one of the following:</p> <ul style="list-style-type: none"> 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. <p>If so, review orange checks at this stage.</p> <ul style="list-style-type: none"> You might want your partially developed code to be free of errors only in the context of the remaining code. <p>If so, do one of the following:</p> <ul style="list-style-type: none"> 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. <p>For more information, see “Provide Context for Verification” on page 16-67.</p>
Later stage or integration stage	<p>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.</p>	<p>Depending on the time you want to spend, do one of the following:</p> <ul style="list-style-type: none"> Review red checks only. Review red and critical orange checks.

Development Stage	Situation	Review Process
Final stage	<ul style="list-style-type: none"> You have provided all your source files. You have emulated run-time environment accurately through the verification options. 	<p>Depending on the time you want to spend, do one of the following:</p> <ul style="list-style-type: none"> Review red checks and critical orange checks. Review red checks and all orange checks. <p>For each orange check:</p> <ul style="list-style-type: none"> 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. <p>As part of your final release process, you can have one of these criteria:</p> <ul style="list-style-type: none"> All red and critical orange checks must be reviewed and justified. All red and orange checks must be reviewed and justified. <p>To justify a check, assign the Status of No action planned or Justified to the check.</p>

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” on page 16-57

More About

- “Orange Checks in Code Prover” on page 16-48

Critical Orange Checks

The software identifies a subset of orange checks that are most likely run-time errors. If you select **Critical checks** from the drop-down list in the left of the **Results List** pane toolbar, you can view this subset.

These 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.

Path

The following example shows a path-related orange check that might be identified as a potential run-time error.

Consider the following code.

```
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 ZDV 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.

Polyspace indicates the definite division by zero error through a **Non-terminating call** error on `path(10)`. If you select the red check on `path(10)`, the **Result Details** pane provides the following information:

```
NTC .... Reason for the NTC: {path.x=10}
```

Bounded Input Values

Most input values can be bounded by data range specifications (DRS). The following example shows an orange check related to bounded input values that might be identified as a potential run-time error.

```
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 for 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:

```
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

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:

```
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: [-231 .. 231-1]
```

Limit Display of Orange Checks

This example shows how to control the number and type of orange checks displayed on the **Results List** pane. 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” on page 16-55.

- 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, SQ0-4, SQ0-5 and SQ0-6. Select an option to see the limit values.

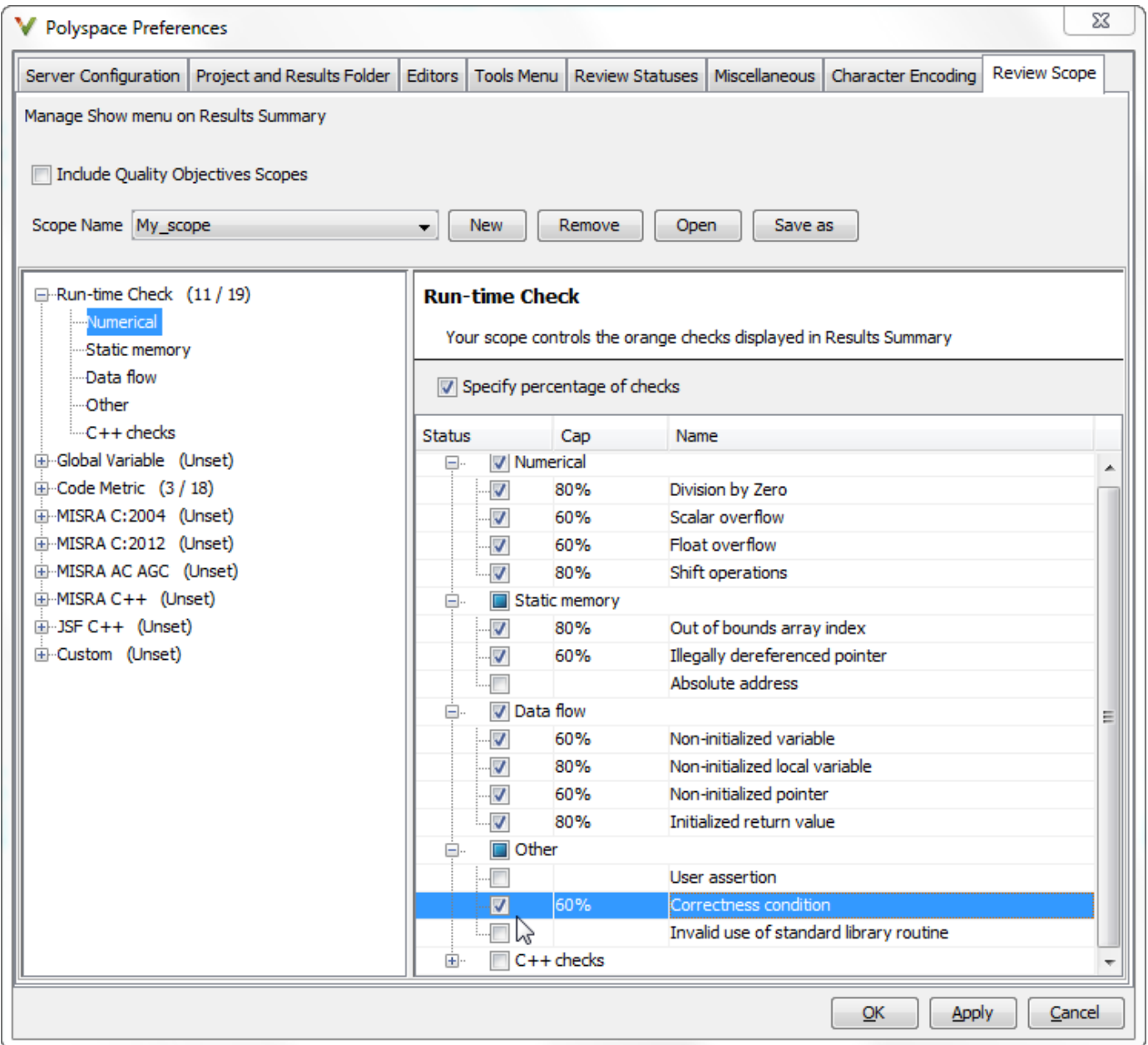
In addition to orange checks, the options impose limits on the display of code metrics and coding rule violations. The option HIS displays code metrics only. For a detailed explanation of the predefined options, see “Software Quality Objectives” on page 16-60.

- 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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

See Also

Related Examples

- “Filter and Group Results” on page 19-2
- “Reduce Orange Checks” on page 16-67
- “Critical Orange Checks” on page 16-55

Software Quality Objectives

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 level or define your own SQOs. 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 <code>goto</code> statements	0
Cyclomatic complexity	10
Number of calling functions	5
Number of calls	7
Number of parameters per function	5
Number of instructions per function	50
Number of call levels in a function	4
Number of <code>return</code> statements in a function	1
Language scope, an indicator of the cost of maintaining or changing functions. Calculated as follows: $(N1+N2) / (n1+n2)$ <ul style="list-style-type: none"> • <i>n1</i> — Number of different operators • <i>N1</i> — Total number of operators • <i>n2</i> — Number of different operands • <i>N2</i> — Total number of operands 	4
Number of recursions	0
Number of direct recursions	0

Metric	Threshold Value
Number of unjustified violations of the following MISRA C:2004 rules: <ul style="list-style-type: none"> • 5.2 • 8.11, 8.12 • 11.2, 11.3 • 12.12 • 13.3, 13.4, 13.5 • 14.4, 14.7 • 16.1, 16.2, 16.7 • 17.3, 17.4, 17.5, 17.6 • 18.4 • 20.4 	0
Number of unjustified violations of the following MISRA C:2012 rules: <ul style="list-style-type: none"> • 8.8, 8.11, and 8.13 • 11.1, 11.2, 11.4, 11.5, 11.6, and 11.7 • 14.1 and 14.2 • 15.1, 15.2, 15.3, and 15.5 • 17.1 and 17.2 • 18.3, 18.4, 18.5, and 18.6 • 19.2 • 21.3 	0
Number of unjustified violations of the following MISRA C++ rules: <ul style="list-style-type: none"> • 2-10-2 • 3-1-3, 3-3-2, 3-9-3 • 5-0-15, 5-0-18, 5-0-19, 5-2-8, 5-2-9 • 6-2-2, 6-5-1, 6-5-2, 6-5-3, 6-5-4, 6-6-1, 6-6-2, 6-6-4, 6-6-5 • 7-5-1, 7-5-2, 7-5-4 • 8-4-1 • 9-5-1 • 10-1-2, 10-1-3, 10-3-1, 10-3-2, 10-3-3 • 15-0-3, 15-1-3, 15-3-3, 15-3-5, 15-3-6, 15-3-7, 15-4-1, 15-5-1, 15-5-2 • 18-4-1 	0

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: <ul style="list-style-type: none"> • 6.3 • 8.7 • 9.2, 9.3 • 10.3, 10.5 • 11.1, 11.5 • 12.1, 12.2, 12.5, 12.6, 12.9, 12.10 • 13.1, 13.2, 13.6 • 14.8, 14.10 • 15.3 • 16.3, 16.8, 16.9 • 19.4, 19.9, 19.10, 19.11, 19.12 • 20.3 	0
Number of unjustified violations of the following MISRA C:2012 rules: <ul style="list-style-type: none"> • 11.8 • 12.1 and 12.3 • 13.2 and 13.4 • 14.4 • 15.6 and 15.7 • 16.4 and 16.5 • 17.4 • 20.4, 20.6, 20.7, 20.9, and 20.11 	0
Number of unjustified violations of the following MISRA C++ rules: <ul style="list-style-type: none"> • 3-4-1, 3-9-2 • 4-5-1 • 5-0-1, 5-0-2, 5-0-7, 5-0-8, 5-0-9, 5-0-10, 5-0-13, 5-2-1, 5-2-2, 5-2-7, 5-2-11, 5-3-3, 5-2-5, 5-2-6, 5-3-2, 5-18-1 • 6-2-1, 6-3-1, 6-4-2, 6-4-6, 6-5-3 • 8-4-3, 8-4-4, 8-5-2, 8-5-3 • 11-0-1 • 12-1-1, 12-8-2 • 16-0-5, 16-0-6, 16-0-7, 16-2-2, 16-3-1 	0
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	

SQO Exhaustive

In addition to all the requirements of SQO Level 1, this level includes the following thresholds. The thresholds for coding rule violations apply only if you check for coding rule violations.

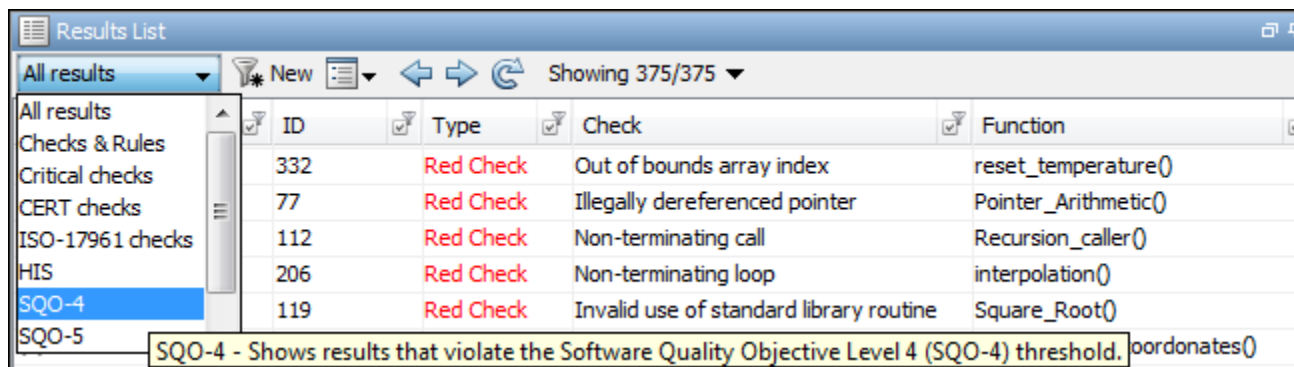
Metric	Threshold Value
Number of unjustified MISRA C and MISRA C++ coding rule violations	0
Number of unjustified red checks	0
Number of unjustified Non-terminating call and Non-terminating loop checks	0
Number of unjustified gray Unreachable code checks	0
Percentage of justified orange checks, calculated 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.

Comparing Verification Results Against Software Quality Objectives

You can compare your verification results against SQOs either in the Polyspace user interface or the Polyspace Metrics web interface.

- 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**.

- In the Polyspace Metrics web interface, you can first determine whether your project fails to attain a certain Software Quality Objective. The web interface generates a **Quality Status** of **PASS** or **FAIL** for your project. If your project has a **Quality Status** of **FAIL**, the web interface highlights in red those results that you must fix or justify to attain the Software Quality Objective. You can choose to only download those results to the Polyspace user interface and review them. For more information, see “Compare Metrics Against Software Quality Objectives” on page 21-15.

You can also generate reports that show the **PASS** or **FAIL** status using the templates `SoftwareQualityObjectives_Summary` and `SoftwareQualityObjectives`. See `Bug Finder` and `Code Prover report (-report-template)`.

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 Metrics web interface, you can compare your project against all predefined SQO levels.

Reduce Orange Checks

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” on page 16-57.

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 “Orange Sources” 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 <code>main</code> 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 10-4

Method	Example
Define execution sequence for multitasking code.	“Configuring Polyspace Multitasking Analysis Manually” on page 11-16
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 (-0)	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.
 - a In the Polyspace user interface, 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	<ul style="list-style-type: none"> • “Software Quality Objective Subsets (C:2004)” on page 12-11 • “Software Quality Objective Subsets (C:2012)” on page 12-18

Type of Code	Option	Rule Description
Generated C code	Check MISRA AC AGC	“Software Quality Objective Subsets (AC AGC)” on page 12-15
Handwritten C++ code	Check MISRA C++ rules	“Software Quality Objective Subsets (C++)” on page 12-21

- b From the option drop-down list, select SQ0-subset1 or SQ0-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, $\{-2,2\} \cup \{10\} \cup \{15,17\}$. 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)*.

See Also

More About

- “Orange Checks in Code Prover” on page 16-48
- “Managing Orange Checks” on page 16-51

Test Orange Checks for Run-Time Errors

This example shows how to run dynamic tests on orange checks. An orange check means that Polyspace static verification detects a possible error but cannot prove it. Orange checks can occur because of:

- Run-time errors.
- Approximations that Polyspace made during static verification.

For more information, see “Orange Checks in Code Prover” on page 16-48.

By running tests, you can determine which orange checks represent run-time errors. Provided that you have emulated the run-time environment accurately, if a dynamic test fails, the orange check represents a run-time error. For this example, save the following code in a file `test_orange.c`:

```
volatile int r;
#include <stdio.h>

int input() {
    int k;
    k = r%21 - 10;
    // k has value in [-10,10]
    return k;
}

void main() {
    int x=input();
    printf("%.2f",1.0/x);
}
```

Run Tests for Full Range of Input

Note The Automatic Orange Tester is not supported on Mac.

- 1 Create a Polyspace project. Add `test_orange.c` to your project.
- 2 In the project configuration, under **Advanced Settings**, select **Automatic Orange Tester**.

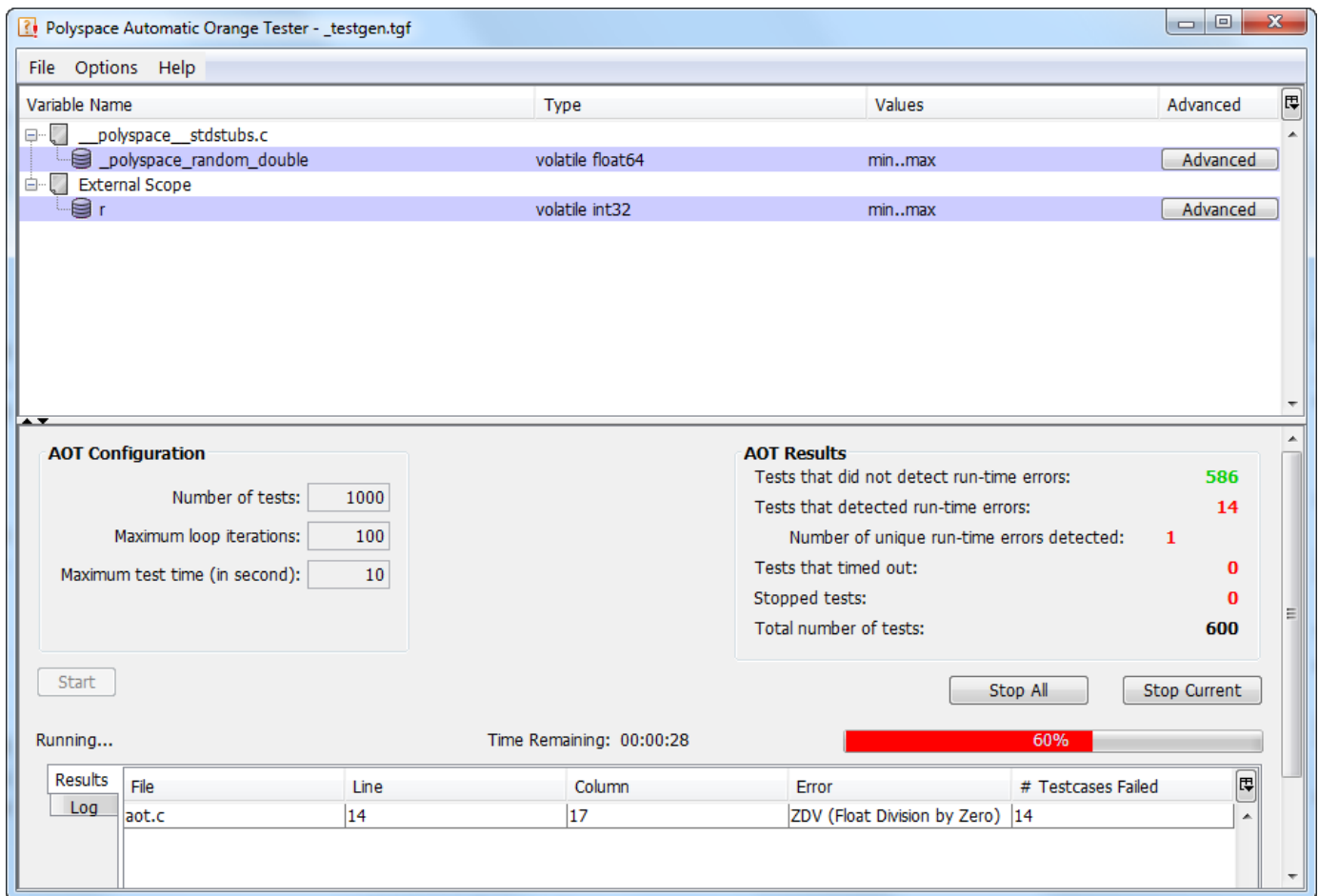
After verification, Polyspace generates additional source code that tests each orange check for run-time errors. The software compiles this instrumented code. When you run the automatic orange tester later, the software tests the resulting binary code.

- 3 Run a verification and open the results.

An orange **Division by zero** error appears on the operation `1.0/x`.

- 4 Select **Tools > Automatic Orange Tester**.
- 5 In the Automatic Orange Tester window, click **Start**.

The Automatic Orange Tester runs tests on your code. If the tests take too long, use the **Stop All** button to stop the tests. Reduce **Number of tests** before running again.



- 6 After the tests are completed, under **AOT Results**, view the number of **Tests that detected run-time errors**.

The orange **Division by zero** check represents a run-time error, so you see test case failures.

- 7 On the **Results** tab, click the row describing the check.

A Test Case Detail window shows:

- The operation on which the tests were run.
- The test cases that failed with the reason for the failure.

Run Tests for Specified Range of Input

The Automatic Orange Tester window lists variables that cause orange checks. Because Polyspace does not have sufficient information about these variables, it makes assumptions about their range. If you know the variable range, you can specify it before running dynamic tests on orange checks. For pointer variables, you can specify the amount of memory written through the pointer. For instance, if the pointer points to an array, you can specify whether the first element of the array or the entire array is written through the pointer.

- 1 In the Automatic Orange Tester window, on the row describing `r`, click **Advanced**.

- 2 In the Edit Values window, under **Variable Values**, select **Range of values**.
- 3 Specify a minimum value of 1 and maximum of 9 for r . The Automatic Orange Tester saves the range as a `.tgf` file in the **Polyspace- Instrumented** folder in your results folder.
- 4 Click **Start** to restart tests on the orange **Division by zero** check for r in $[1, 9]$.

A division by zero cannot occur for r in $[1, 9]$, so there are no test failures. Although a test failure indicates a run-time error for specified inputs, because of the finite number of tests performed, the absence of test failures does not mean absence of a run-time error.

- 5 To prove that your range converts the orange check into a green check, you must provide the range during another static verification.
 - a In the Automatic Orange Tester window, select **File > Export Constraints**.
 - b Save your ranges as a text file.
 - c Before running the next verification, on the **Configuration** pane, under **Inputs & Stubbing**, provide the text file for **Constraint setup**.
 - d Run a verification and open your results.

Instead of orange, there is a green **Division by zero** check on the operation $1.0/x$.

See Also

Related Examples

- “Identify Function Call with Run-Time Error” on page 17-44
- “Limit Display of Orange Checks” on page 16-57

More About

- “Limitations of Automatic Orange Tester” on page 16-73
- “Orange Checks in Code Prover” on page 16-48
- “Managing Orange Checks” on page 16-51

Limitations of Automatic Orange Tester

The Automatic Orange Tester has the following limitations:

Unsupported Platforms

The Automatic Orange Tester is not supported on Mac.

Unsupported Polyspace Options

The software does not support the following options with `-automatic-orange-tester`.

- `-div-round-down`
- `-char-is-16bits`
- `-short-is-8bits`

In addition, the software does not support global asserts in the code of the form `Pst_Global_Assert(A,B)`.

Options with Restrictions

Do not specify the following with `-automatic-orange-tester`:

- `-allow-non-finite-floats`
- `-check-subnormal`
- `-data-range-specification` (in global assert mode)
- `-target [c18 | tms320c3c | x86_64 | sharc21x61]`

You must use the `-target mcpu` option together with `-pointer-is-32bits`.

Unsupported C Routines

The software does not support verification of C code that contains calls to the following routines:

- `va_start`
- `va_arg`
- `va_end`
- `va_copy`
- `setjmp`
- `sigsetjmp`
- `longjmp`
- `siglongjmp`
- `signal`
- `sigset`
- `sighold`

- sigelse
- sigpause
- sigignore
- sigaction
- sigpending
- sigsuspend
- sigvec
- sigblock
- sigsetmask
- sigprocmask
- siginterrupt
- srand
- srandom
- initstate
- setstate

Reviewing Checks

- “Review and Fix Absolute Address Usage Checks” on page 17-2
- “Review and Fix Correctness Condition Checks” on page 17-3
- “Review and Fix Division by Zero Checks” on page 17-7
- “Review and Fix Function Not Called Checks” on page 17-11
- “Review and Fix Function Not Reachable Checks” on page 17-13
- “Review and Fix Function Not Returning Value Checks” on page 17-15
- “Review and Fix Illegally Dereferenced Pointer Checks” on page 17-16
- “Review and Fix Incorrect Object Oriented Programming Checks” on page 17-22
- “Review and Fix Invalid C++ Specific Operations Checks” on page 17-24
- “Review and Fix Invalid Shift Operations Checks” on page 17-26
- “Review and Fix Invalid Use of Standard Library Routine Checks” on page 17-30
- “Invalid Use of Standard Library Floating Point Routines” on page 17-32
- “Review and Fix Non-initialized Local Variable Checks” on page 17-35
- “Review and Fix Non-initialized Pointer Checks” on page 17-38
- “Review and Fix Non-initialized Variable Checks” on page 17-40
- “Review and Fix Non-Terminating Call Checks” on page 17-42
- “Identify Function Call with Run-Time Error” on page 17-44
- “Review and Fix Non-Terminating Loop Checks” on page 17-46
- “Identify Loop Operation with Run-Time Error” on page 17-49
- “Review and Fix Null This-pointer Calling Method Checks” on page 17-51
- “Review and Fix Out of Bounds Array Index Checks” on page 17-53
- “Review and Fix Overflow Checks” on page 17-57
- “Detect Overflows in Buffer Size Computation” on page 17-61
- “Review and Fix Return Value Not Initialized Checks” on page 17-63
- “Review and Fix Uncaught Exception Checks” on page 17-66
- “Review and Fix Unreachable Code Checks” on page 17-68
- “Review and Fix User Assertion Checks” on page 17-72
- “Find Relations Between Variables in Code” on page 17-75
- “Review Polyspace Results on AUTOSAR Code” on page 17-78

Review and Fix Absolute Address Usage Checks

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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

Review and Fix Correctness Condition Checks

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.

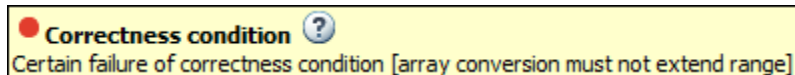
For the general workflow that applies to all checks, see “Interpret Polyspace Code Prover Results” on page 16-2.

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:

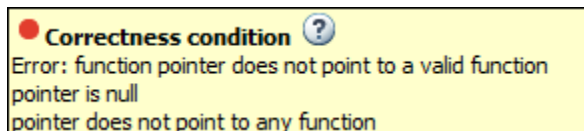
- An array is converted to another array of larger size.

In the following example, a red check occurs because an array is converted to another array of larger size.



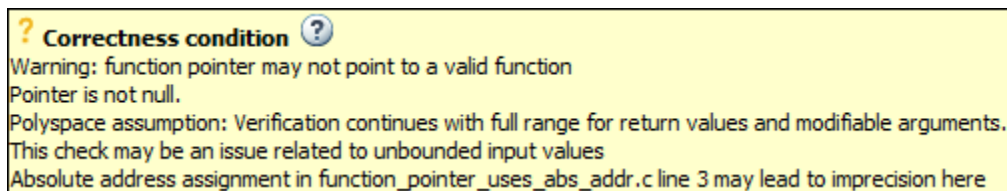
- When dereferenced, a function pointer has value NULL.

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`.

? 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 returned value (function returns type int 32 but call expects type float 64).

Polyspace assumption: Verification continues with full range for return values and modifiable arguments.

- The value of a variable falls outside the range that you specify through the **Global Assert** mode. See “Constrain Global Variable Range” on page 10-13.

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.

● Correctness condition ?

Certain failure of global assertion condition [glob in the range of 0...10]

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.	<ol style="list-style-type: none"> 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. <ol style="list-style-type: none"> 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.

Check Information	How to Determine Root Cause
<p>Issues when dereferencing a function pointer:</p> <ul style="list-style-type: none"> • 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. 	<ol style="list-style-type: none"> 1 Find the location where you assign the function pointer to a function. <ol style="list-style-type: none"> 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> <ol style="list-style-type: none"> 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.
<p>The value of a variable falls outside the range that you specify through the Global Assert mode.</p>	<p>Browse through all previous instances of the global variable. Identify a suitable point to constrain the variable.</p> <ol style="list-style-type: none"> 1 Right-click the variable. Select Show In Variable Access View. 2 On the Variable Access pane, select each instance of the variable.

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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

Review and Fix Division by Zero Checks

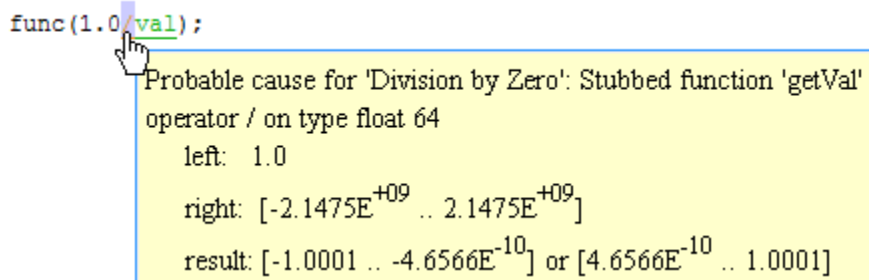
Follow one or more of these steps until you determine a fix for the **Division by zero** check. There are multiple ways to fix a red or orange check. For a description of the check and code examples, see [Division by zero](#).

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 Polyspace Code Prover Results” on page 16-2.

Step 1: Interpret Check Information

Place your cursor on the / or % operation that causes the **Division by zero** error.



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
<pre>if(val != 0) func(1.0/val); else /* Error handling */</pre>	<pre>#define eps 0.0000001 . . if(val < -eps val > eps) func(1.0/val); else /* Error handling */</pre>

- 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 “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.

Variable	How to Find Previous Instances of Variable
Local Variable	<p>Use one of the following methods:</p> <ul style="list-style-type: none"> • Search for the variable. <ol style="list-style-type: none"> 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. <ol style="list-style-type: none"> 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.	<ol style="list-style-type: none"> 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 <code>ret=func();</code>	<ol style="list-style-type: none"> 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 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 17-75.

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” on page 10-2.

- 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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

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

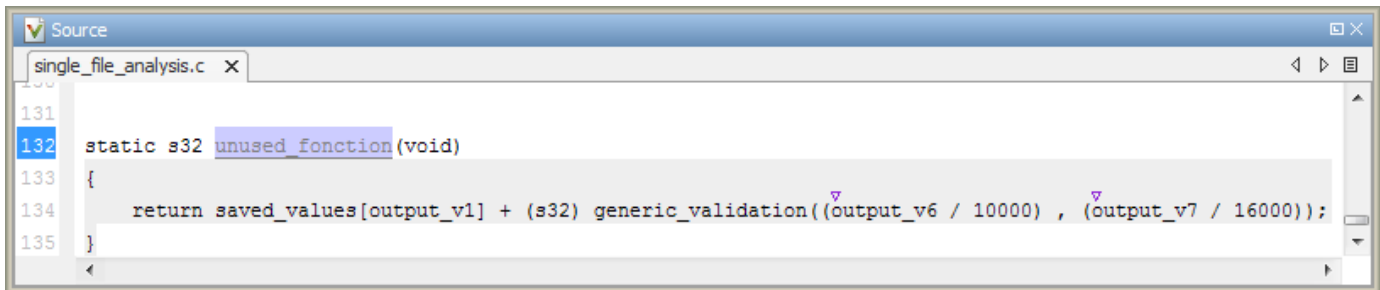
Follow one or more of these steps until you determine a fix for the **Function not called** check. There are multiple ways to fix this check. For a description of the check and code examples, see [Function not called](#).

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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

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 x
131
132 static s32 unused_function(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 11-16.

Review and Fix Function Not Reachable Checks

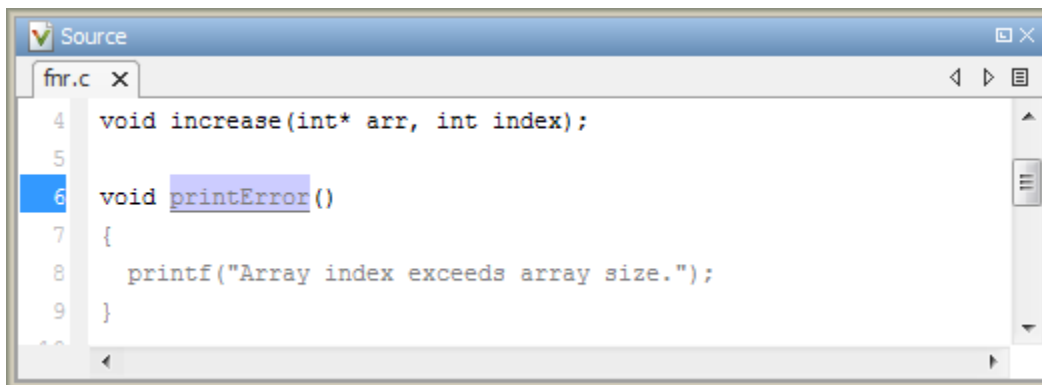
Follow one or more of these steps until you determine a fix for the **Function not reachable** check. There are multiple ways to fix this check. For a description of the check and code examples, see [Function not reachable](#).

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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

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  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 17-68.

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**.

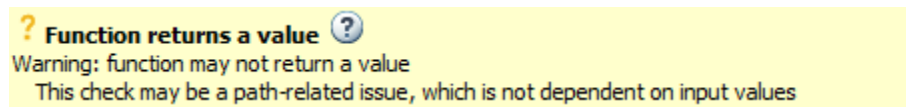
Review and Fix Function Not Returning Value Checks

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 Polyspace Code Prover Results” on page 16-2.

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

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 Polyspace Code Prover Results” on page 16-2.

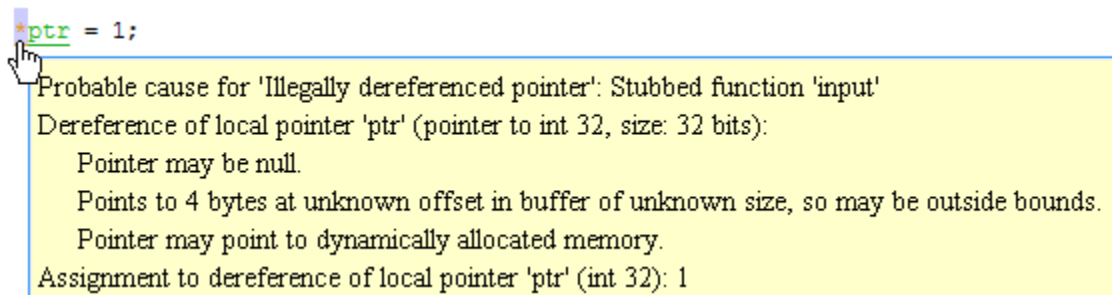
Step 1: Interpret Check Information

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.



Possible fix: Dereference `ptr` only if it is not NULL.

```
if(ptr !=NULL)
    *ptr = 1;
else
    /* Alternate action */
```

- Whether the pointer points to dynamically allocated memory.

In the following example, `ptr` can point to dynamically allocated memory. It is possible that the dynamic memory allocation operator returns NULL.

```
*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: Check the return value of the memory allocation operator for NULL.

```
ptr = (char*) malloc(i);
if(ptr==NULL)
  /* Error handling*/
else {
  .
  .
  *ptr=0;
  .
  .
}
```

- Whether pointer points outside allowed bounds. A pointer points outside bounds 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 allowed buffer.

- Whether pointer can point outside allowed bounds 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`. For instance, specify that `getAddress` returns a pointer to a 10-element array. For more information, see “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.	<p>Find an execution path where the pointer is assigned the value NULL or not assigned a definite address.</p> <ol style="list-style-type: none"> 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. <i>Possible fix:</i> If the pointer can be NULL, place a check for NULL immediately after the assignment. <pre> if(ptr==NULL) /* Error handling*/ else { . . } </pre> 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. <i>Possible fix:</i> Assign a valid address to the pointer in all branches of the conditional statement.
Pointer can point to dynamically allocated memory.	<p>Identify where the allocation occurs.</p> <ol style="list-style-type: none"> 1 Right-click the pointer and select Search For All References. 2 Find the previous instance where the pointer receives a value from a dynamic memory allocation function such as <code>malloc</code>. <i>Possible fix:</i> After the allocation, test the pointer for NULL.

Error	How to Find Root Cause
<p>Pointer can point outside bounds allowed by the buffer.</p>	<p>1 Find the allowed buffer.</p> <ul style="list-style-type: none"> 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. <p>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.</p> <p>2 Find out why the pointer points outside the allowed buffer.</p> <ul style="list-style-type: none"> 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. <p><i>Possible fix:</i> Remove unintended pointer arithmetic. To avoid pointer arithmetic that takes a pointer outside allowed 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 allowed buffer.</p>

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 Coding Standard Violations” on page 12-2.

- 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 17-44.

- 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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

For instance, 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 an allowed buffer size based on the type of the pointer.

If you increment the pointer, you exceed the allowed 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 allowed 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 17-22

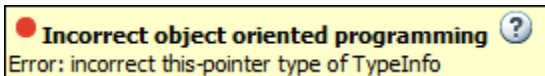
“Step 2: Determine Root Cause of Check” on page 17-22

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 Polyspace Code Prover Results” on page 16-2.

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.

Error	How to Find Root Cause
You dereference a function pointer that points to an invalid member function.	<p>Compare the argument and return types of the function pointer and the member function that it points to.</p> <ol style="list-style-type: none"> Right-click the function pointer on the Source pane and select Search For All References. Find the instances where you: <ul style="list-style-type: none"> Define the function pointer. Assign the address of a member function to the function pointer. Find the member function definition. Right-click the member function name on the Source pane and select Go To Definition.
You call a pure <code>virtual</code> member function from a constructor or destructor.	<p>Find the member function declaration and determine whether you intended to declare it as <code>virtual</code> or <code>pure virtual</code>. Alternatively, determine if you can replace the call to the <code>pure virtual</code> function with another operation, for instance, a call to a different member function.</p> <ol style="list-style-type: none"> Right-click the function name on the Source pane and select Search for <code>function_name</code> in All Source Files. Find the function declaration from the search results. <p>A <code>pure virtual</code> function has a declaration such as:</p> <pre>virtual void func() = 0;</pre>
You call a member function using an incorrect <code>this</code> pointer.	<p>Determine why the <code>this</code> pointer is incorrect.</p> <p>For instance, if a red Incorrect object oriented programming check appears on a function call <code>ptr->func()</code> and the message indicates that the <code>this</code> pointer is incorrect, trace the data flow for <code>ptr</code>.</p> <ul style="list-style-type: none"> 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: <ul style="list-style-type: none"> Cast between pointers of unrelated types. Pointer arithmetic that takes a pointer outside its allowed buffer, for instance, the bounds of an array. <p>If a red Incorrect object oriented programming check appears on a function call <code>obj.func()</code>, trace the data flow for <code>obj</code>. See if <code>obj</code> is not initialized previously.</p>

Review and Fix Invalid C++ Specific Operations Checks

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 Polyspace Code Prover Results” on page 16-2.

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.	<ol style="list-style-type: none"> <li data-bbox="578 300 1484 447">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 17-7. <li data-bbox="578 447 1484 520">2 Identify a point where you can constrain the array size variable to positive values.
The typeid operator dereferences a possibly NULL pointer.	<ol style="list-style-type: none"> <li data-bbox="578 541 1484 688">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 17-16. <li data-bbox="578 688 1484 730">2 Identify a point where you can test the pointer for NULL.
The dynamic_cast operator performs an invalid cast.	<p data-bbox="578 751 1484 804">Navigate to the definitions of the classes involved. Determine the inheritance relationship between the classes.</p> <ol style="list-style-type: none"> <li data-bbox="578 835 1484 888">1 On the Source pane in the Polyspace user interface, right-click the class name. <li data-bbox="578 909 1484 932">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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

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

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 Polyspace Code Prover Results” on page 16-2.

Step 1: Interpret Check Information

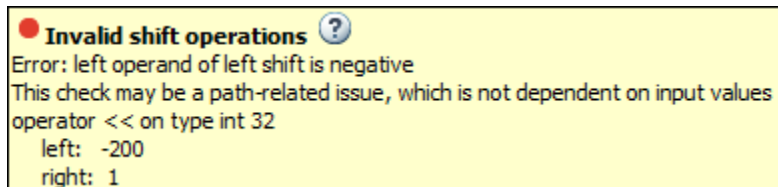
Select the red or orange **Invalid shift operations** check. Obtain the following information from the **Result Details** pane:

- 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.

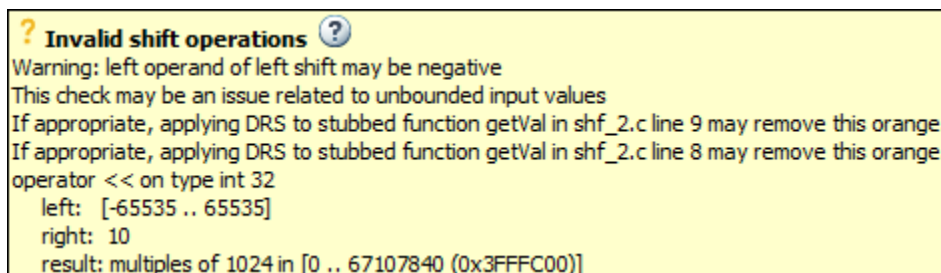
In the example below, a red error occurs because the shift amount is outside allowed bounds. The allowed range for the shift amount is 0 to 31.



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.



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 “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 “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 “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	<p>Use one of the following methods:</p> <ul style="list-style-type: none"> • Search for the variable. <ol style="list-style-type: none"> 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. <ol style="list-style-type: none"> 1 Double-click the variable on the Source pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
<p>Global Variable</p> <p>Right-click the variable. If the option Show In Variable Access View appears, the variable is a global variable.</p>	<ol style="list-style-type: none"> 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 .
<p>Function return value</p> <pre>ret=func();</pre>	<ol style="list-style-type: none"> 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 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 17-75.

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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

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

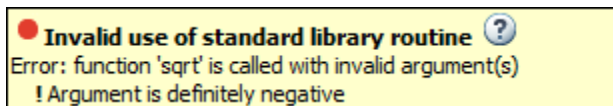
Follow one or more of these steps until you determine a fix for the **Invalid use of standard library routine** check. For a description of the check and code examples, see [Invalid use of standard library routine](#).

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 Polyspace Code Prover Results” on page 16-2.

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.

Function	Cause of Red or Orange Check
islower, isdigit, and other character-handling functions in ctype.h	The value of the argument can be outside the range allowed for an unsigned char variable. Note that you can use the macro EOF as argument.
Functions in math.h	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 17-32.
sqrt	The value of the argument can be negative.
pow	The first argument can be negative while the second argument is a non-integer.
exp, exp2, or the hyperbolic functions	The argument can be so large that the result exceeds the value allowed for a double.
log	The argument can be zero or negative.

Function	Cause of Red or Orange Check	
	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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

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 “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

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 `frexpl` 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

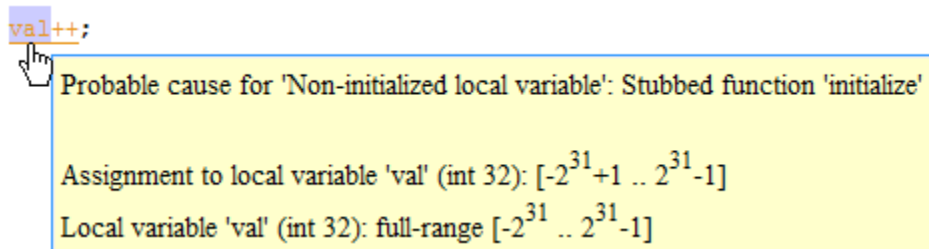
Follow one or more of these steps until you determine a fix for the **Non-initialized local variable** check. There are multiple ways to fix this check. For a description of the check and code examples, see [Non-initialized local variable](#).

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 Polyspace Code Prover Results” on page 16-2.

Step 1: Interpret Check Information

Place your cursor on the variable on which the **Non-initialized local variable** error appears.



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 “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 17-68.

- 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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

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

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 Polyspace Code Prover Results” on page 16-2.

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 17-68.
 - 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: <ul style="list-style-type: none"> • Search for the variable. <ol style="list-style-type: none"> 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. <ol style="list-style-type: none"> 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.	<ol style="list-style-type: none"> 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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

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

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 Polyspace Code Prover Results” on page 16-2.

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 17-68.

- 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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

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

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 Polyspace Code Prover Results” on page 16-2.

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 17-44.

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.

Division by Zero 

Warning (probable error): scalar division by zero may occur
operator / on type int 32
left: full-range $[-2^{31} .. 2^{31}-1]$
right: full-range $[-2^{31} .. 2^{31}-1]$
result: full-range $[-2^{31} .. 2^{31}-1]$

Calling context	File	Scope	Line
operator / on type int 32 left: 1 right: 0	file.c	main	9
operator / on type int 32 left: 1 right: 11 result: 0	file.c	main	8

See Also

Non-terminating call

Related Examples

- “Review and Fix Non-Terminating Call Checks” on page 17-42
- “Test Orange Checks for Run-Time Errors” on page 16-70

More About

- “Orange Checks in Code Prover” on page 16-48

Review and Fix Non-Terminating Loop Checks

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 Polyspace Code Prover Results” on page 16-2.

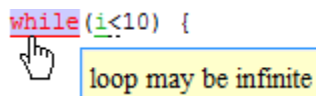
Step 1: Interpret Check Information

Place your cursor on the loop keyword such as `for` or `while`.

Obtain the following information from the tooltip:

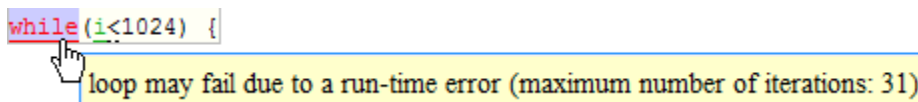
- Whether the loop is infinite or contains a run-time error.

In the following example, it is likely that the loop is infinite.



- 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.



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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

- 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 17-49.

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.

● Non-terminating loop ? The loop is infinite or contains a run-time error. This check may be a path-related issue, which is not dependent on input values Loop fails due to a run-time error (maximum number of iterations: 10).				
	Event	File	Scope	Line
1	Iterating on loop: loop ran 9 times	file.c	main()	5
2	Entering function 'f'	file.c	main()	6
3	Iterating on loop: loop ran 10 times	file.c	f()	13
4	Array index is outside its bounds : [0..99]	file.c	f()	14
5	● The loop is infinite or contains a run-time error.	file.c	main()	5

You can read the event history in sequence. The outer 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 17-46
- “Test Orange Checks for Run-Time Errors” on page 16-70

More About

- “Orange Checks in Code Prover” on page 16-48

Review and Fix Null This-pointer Calling Method Checks

In this section...

“Step 1: Interpret Check Information” on page 17-51

“Step 2: Determine Root Cause of Check” on page 17-51

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 Polyspace Code Prover Results” on page 16-2.

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

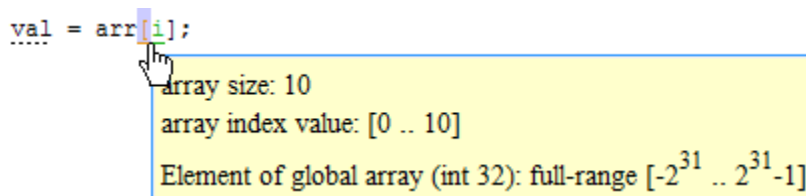
Follow one or more of these steps until you determine a fix for the **Out of bounds array index** check. There are multiple ways to fix the check. For a description of the check and code examples, see [Out of bounds array index](#).

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 Polyspace Code Prover Results” on page 16-2.

Step 1: Interpret Check Information

Place your cursor on the [symbol.



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: <ul style="list-style-type: none"> • Search for the variable. <ol style="list-style-type: none"> 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. <ol style="list-style-type: none"> 1 Double-click the variable on the Source pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.

Variable	How to Find Previous Instances of Variable
Global Variable Right-click the variable. If the option Show In Variable Access View appears, the variable is a global variable.	<ol style="list-style-type: none"> 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 <pre>ret=func();</pre>	<ol style="list-style-type: none"> 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.

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 17-75.

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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

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

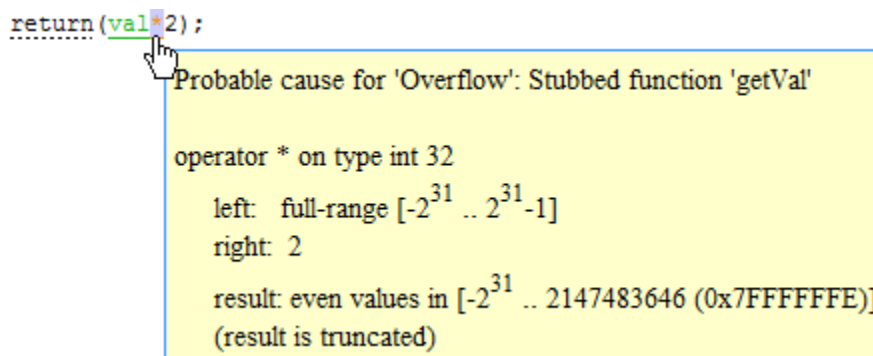
Follow one or more of these steps until you determine a fix for the **Overflow** check. There are multiple ways to fix the check. For a description of the check and code examples, see [Overflow](#).

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 Polyspace Code Prover Results” on page 16-2.

Step 1: Interpret Check Information

Place your cursor on the operation that overflows.



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 “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 “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	<p>Use one of the following methods:</p> <ul style="list-style-type: none"> • Search for the variable. <ol style="list-style-type: none"> 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. <ol style="list-style-type: none"> 1 Double-click the variable on the Source pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
<p>Global Variable</p> <p>Right-click the variable. If the option Show In Variable Access View appears, the variable is a global variable.</p>	<ol style="list-style-type: none"> 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 ▶.
<p>Function return value</p> <pre>ret=func();</pre>	<ol style="list-style-type: none"> 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 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 17-75.

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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

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

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 Polyspace Code Prover Results” on page 16-2.

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 “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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

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

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 <code>catch</code> block. The exception escapes to the <code>main</code> .
Call to <i>typeName</i> throws during "catch" parameter construction.	Creating the <code>catch</code> 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`

Uncaught exception ?

Error: unhandled exception propagates to main or entry-point function

	Event	File	Scope
1	Exception thrown	excp.cpp	initialVector::getValue(int)
2	Exiting function 'initialVector::getValue(int)'	excp.cpp	getValueFromVector(initialVector *)
3	Exiting function 'getValueFromVector(initialVector *)'	excp.cpp	main
4	Error: unhandled exception propagates to main or entry-point function	excp.cpp	main()

Configuration Result Details

Source

excp.cpp X

```

24
25 int initialVector::getValue(int index) {
26     if(index >= 0 && index < sizeVector)
27         return table[index];
28     else throw error();
29 }
30
31 int getValueFromVector(initialVector* vectorPtr) {
32     return vectorPtr->getValue(5);
33 }
34
35 void main() {
36     initialVector *vectorPtr = new initialVector(5);
37     int aVal = getValueFromVector(vectorPtr);
38     }
```

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

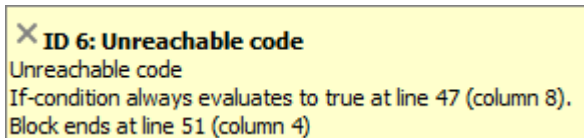
Follow one or more of these steps until you determine a fix for the **Unreachable code** check. There are multiple ways to fix this check. For a description of the check and code examples, see [Unreachable code](#).

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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

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.



- 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 17-70.

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	<p>Use one of the following methods:</p> <ul style="list-style-type: none"> • Search for the variable. <ol style="list-style-type: none"> 1 Right-click the variable. Select Search For All References. <p>All instances of the variable appear on the Search pane with the current instance highlighted.</p> <ol style="list-style-type: none"> 2 On the Search pane, select the previous instances. • Browse the source code. <ol style="list-style-type: none"> 1 Double-click the variable on the Source pane. <p>All instances of the variable are highlighted.</p> <ol style="list-style-type: none"> 2 Scroll up to find the previous instances.

Variable	How to Find Previous Instances of Variable
Global Variable Right-click the variable. If the option Show In Variable Access View appears, the variable is a global variable.	<ol style="list-style-type: none"> 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 <pre>ret=func();</pre>	<ol style="list-style-type: none"> 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.

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 17-75.

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 16-43

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

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 Polyspace Code Prover Results” on page 16-2.

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 17-70.

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	<p>Use one of the following methods:</p> <ul style="list-style-type: none"> • Search for the variable. <ol style="list-style-type: none"> 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. <ol style="list-style-type: none"> 1 Double-click the variable on the Source pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
<p>Global Variable</p> <p>Right-click the variable. If the option Show In Variable Access View appears, the variable is a global variable.</p>	<ol style="list-style-type: none"> 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 .
<p>Function return value</p> <pre>ret=func();</pre>	<ol style="list-style-type: none"> 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 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 17-75.

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 17-44.
- 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 Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

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.

For a tutorial, see “Test Orange Checks for Run-Time Errors” on page 16-70.

- 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. These steps can be performed in projects using the C language only.

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 Set up the Polyspace analysis configuration:
 - Set the source code language to C. Use the analysis option `Source code language (-lang)`.
 - 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`. 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 Polyspace Code Prover Results” on page 16-2

Review Polyspace Results on AUTOSAR Code

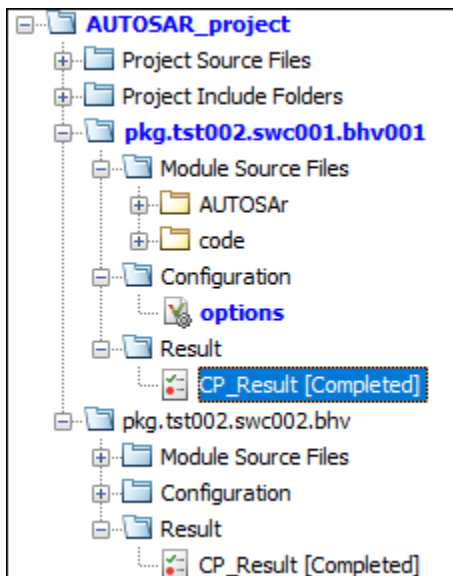
This tutorial describes how to open Polyspace Code Prover results for AUTOSAR-specific code and interpret results that highlight violation of data constraints in the ARXML.

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 7-12.

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 17-79.
- 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 Web Interface” (Polyspace Code Prover Server) and “Interpret Results” (Polyspace Code Prover Access).

Use the first method for easier understanding of results.

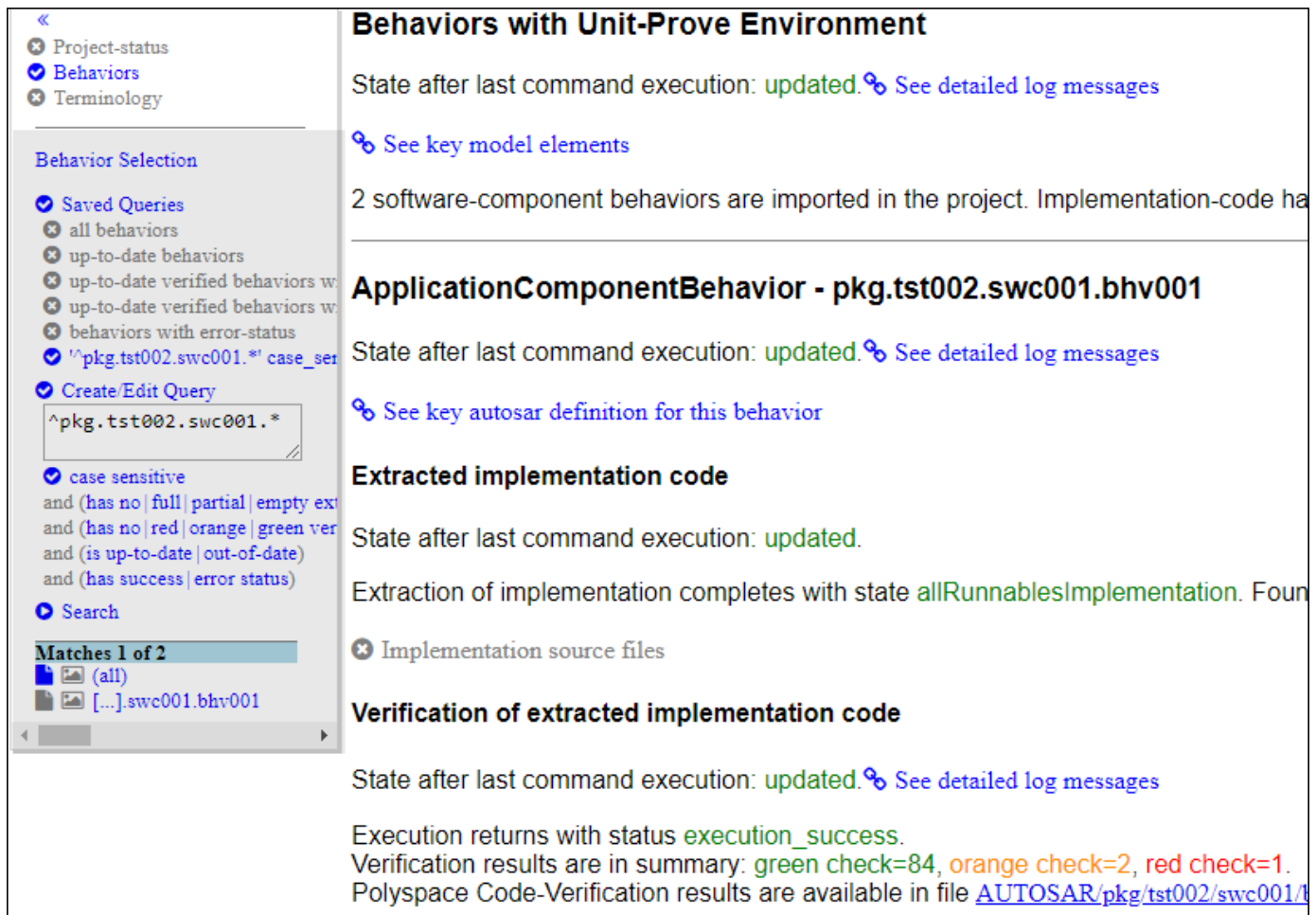
In the file `psar_project.xhtml`, click the  icon on the upper left. On the left pane, click **Behaviors**. You can see the list of all software components whose internal behavior-s are extracted.

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.



Behaviors with Unit-Prove Environment

State after last command execution: **updated**. [See detailed log messages](#)

[See key model elements](#)

2 software-component behaviors are imported in the project. Implementation-code ha

ApplicationComponentBehavior - pkg.tst002.swc001.bhv001

State after last command execution: **updated**. [See detailed log messages](#)

[See key autosar definition for this behavior](#)

Extracted implementation code

State after last command execution: **updated**.

Extraction of implementation completes with state **allRunnablesImplementation**. Four

Implementation source files

Verification of extracted implementation code

State after last command execution: **updated**. [See detailed log messages](#)

Execution returns with status **execution_success**.

Verification results are in summary: **green check=84, orange check=2, red check=1**.

Polyspace Code-Verification results are available in file [AUTOSAR/pkg/tst002/swc001/t](#)

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-implementation
- behaviors with error-status

Edit Query

.*

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

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

Result Details

? Invalid use of AUTOSAR runtime environment function 

Warning: Function 'Rte_IWrite_step_out_e4' is called with possibly invalid argument(s)

- Conditions on first argument 'self' (see [parameter spec](#)):
 - ✓ self meets its specification.
Specification: non-NULL
 - ✓ self meets its specification.
Specification: allocated
 - ✓ self->Rte_Dummy meets its specification.
Specification: [0 .. 255]
Actual value (unsigned int 8): full-range [0 .. 255]
- Conditions on second argument 'aData' (see [parameter spec](#)):
 - ✓ aData meets its specification.
Specification: non-NULL
 - ✓ aData meets its specification.
Specification: allocated
 - ? aData[] may not meet its specification.
Specification: [-320 .. 320]
Actual value (int 32): [-320 .. 321]

AUTOSAR Specification

Focus on: One Function Specific Parameter

Rte_IWrite_step_out_e4

Function required by Autosar Software-Component

▼ signature

[2] IN aData is a app_Array_2_n320to320ConstRef constant pointer to a constant Matrix application type pkg.types.app.Array_2_n320to320 1D-Matrix [2] of Integer application type pkg.types.app.Int_n320to320
Values must be in constrained-range [-320 .. 320]

▼ Base software-type

▼ Physical-range

Source

```
// read array
app_Array_2_n320to320ConstRef e4;
e4 = Rte_IRead_step_in_e4(self);
if (e4!=NULL_PTR) {
    // write possibly invalid array element
    app_Array_2_n320to320 const e4b = {e4[0]+1,e4[1]};
    Rte_IWrite_step_out_e4(self, (se4b));
}
```

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 Polyspace Code Prover Results” on page 16-2.

Decide whether to fix your code or ARXML, or justify the result through comments. See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

See Also

Invalid result of AUTOSAR runnable implementation | Invalid use of AUTOSAR runtime environment function

More About

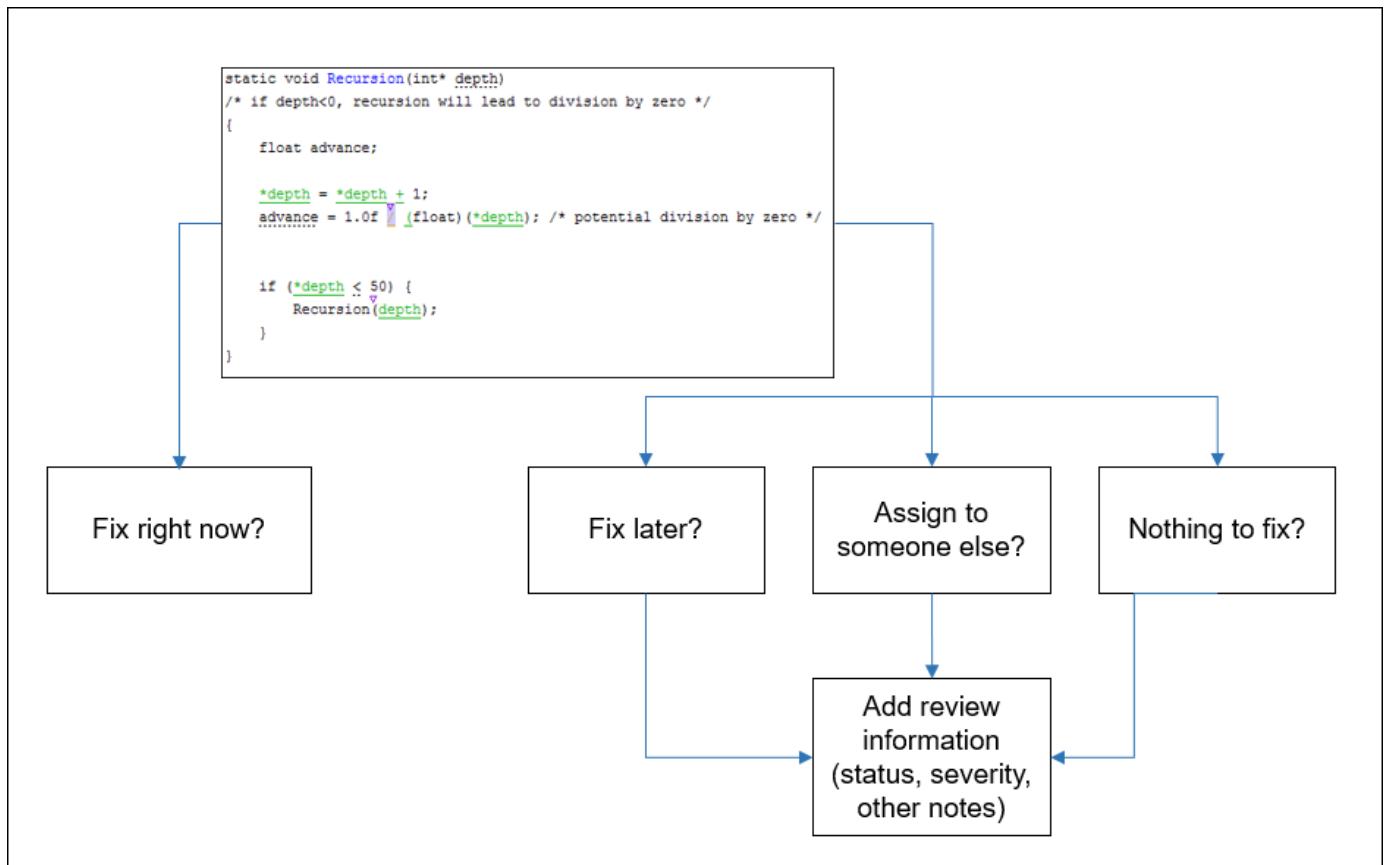
- “Benefits of Polyspace for AUTOSAR” on page 7-5
- “Run Polyspace on AUTOSAR Code” on page 7-12

Fix or Comment Polyspace Results

- “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2
- “Annotate Code and Hide Known or Acceptable Results” on page 18-6
- “Short Names of Code Prover Run-Time Checks” on page 18-12
- “Short Names of Code Complexity Metrics” on page 18-14
- “Annotate Code for Known or Acceptable Results (Not Recommended)” on page 18-16
- “Define Custom Annotation Format” on page 18-20
- “Annotation Description Full XML Template” on page 18-27
- “Justify Coding Rule Violations Using Code Prover Checks” on page 18-33

Address Polyspace Results Through Bug Fixes or Justifications

Once you understand the root cause of a Polyspace finding, you can fix your code. Otherwise, add review information to your Polyspace results to fix the code later or to justify the result. You can use the information to keep track of your review progress.



If you add review information to your results file, they carry over to the results of the next analysis on the same project. If you add the same information as comments to your code (annotate), they carry over to any subsequent analysis of the code, whether in the same project or not. You can also hide results using code annotations.

Add Review Information to Results File

The screenshot shows the 'Result Details' pane in Polyspace. At the top, there is a 'Variable trace' checkbox which is unchecked. Below it is the 'Result Review' section, which includes 'Severity' set to 'High' and 'Status' set to 'Fix'. A text box contains the note 'Adding missing else condition.'. Below this is a yellow warning banner for a 'Non-initialized pointer' (Impact: High) with a question mark icon. The description reads: 'Local pointer 'pi' may be read before being initialized.'. At the bottom is a table with the following data:

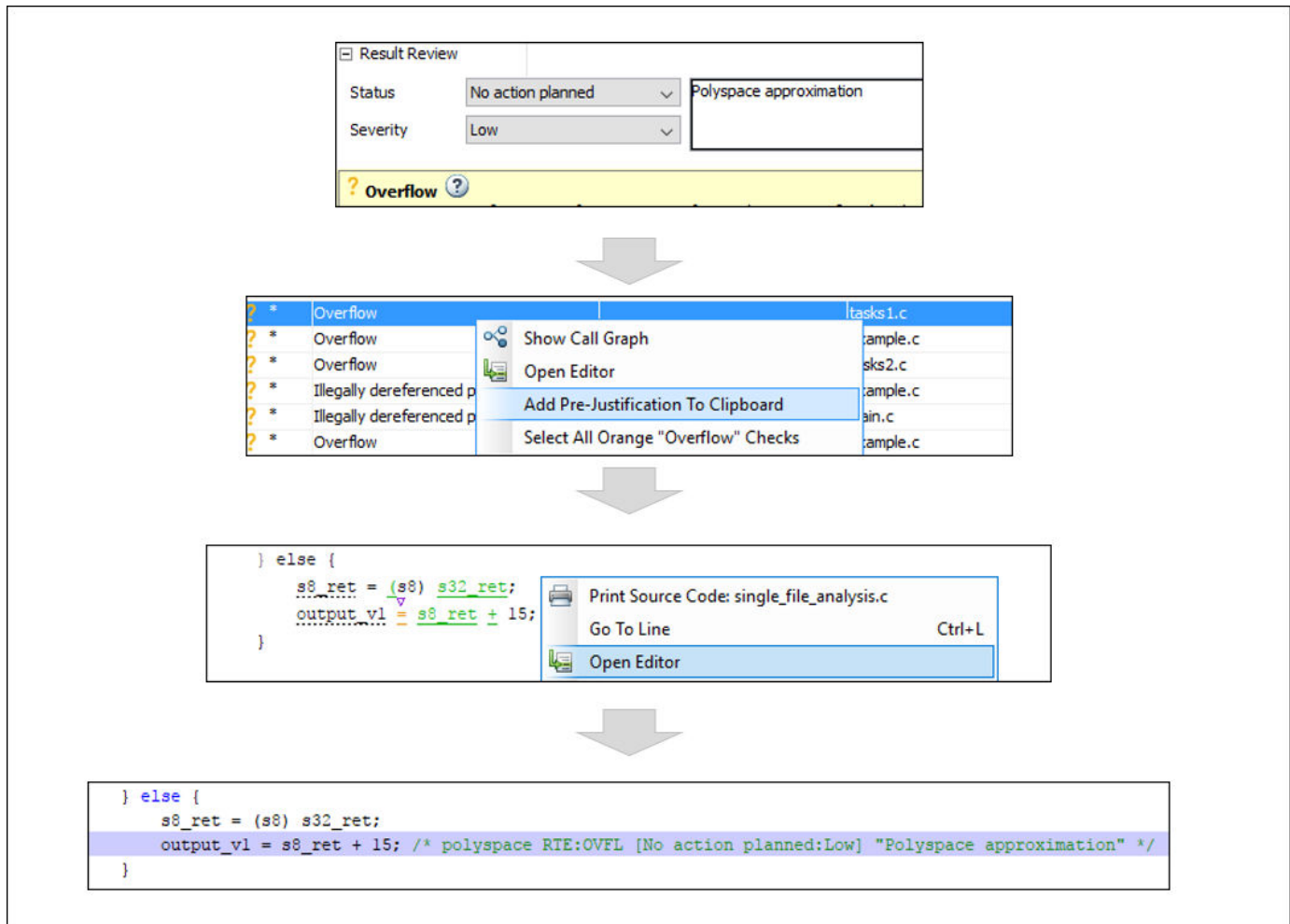
	Event	File	Scope	Line
1	Declaration of variable 'pi'	dataflow.c	bug_noninitptr()	152
2	Not entering if statement (if-condition false)	dataflow.c	bug_noninitptr()	154
3	Non-initialized pointer	dataflow.c	bug_noninitptr()	159

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

- Justified
- No Action Planned
- Not a Defect

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

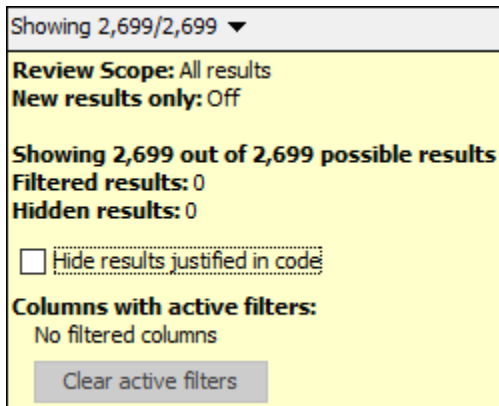
Comment or Annotate in Code



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

You can either type the annotation directly or copy it from the user interface. In the user interface, to copy annotations, right-click a result and select **Add Pre-Justification To Clipboard**. Open your source code in an editor and paste *on the same line as* the result. If you follow this workflow, Polyspace assumes that you have set a status of **No Action Planned**. The software hides the result from all places (except reports needed for certification). The only exceptions are the safety-critical Code Prover run-time checks, which are hidden from the results list but not the source code.

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



Showing 2,699/2,699 ▼

Review Scope: All results
New results only: Off

Showing 2,699 out of 2,699 possible results
Filtered results: 0
Hidden results: 0

Hide results justified in code

Columns with active filters:
No filtered columns

Clear active filters

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

If you want to directly type the annotation, see the annotation syntax in “Annotate Code and Hide Known or Acceptable Results” on page 18-6.

See Also

More About

- “Annotate Code and Hide Known or Acceptable Results” on page 18-6
- “Import Review Information from Previous Polyspace Analysis” on page 15-2

Annotate Code and Hide Known or Acceptable Results

If a Polyspace analysis of your code finds known or acceptable defects or coding rule violations, you can suppress the defects or violations in subsequent analyses. Add code annotations indicating that you have reviewed the issues and that you do not intend to fix them.

You can add annotations through the Polyspace user interface or by typing them directly in your code. For the general workflow, see “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2. This topic shows the annotation syntax.

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.

- Annotation for current line of code (including within macros):

```
line of code; /* polyspace Family:Result_name */
```

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

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

- Annotation for block of code:

```
/* polyspace-begin Family:Result_name */
code;
/* polyspace-end Family:Result_name */
```

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

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

When you annotate a block of code, if subsequent annotations nested within that block of code apply to the same `Family` and `Result_name`, the nested annotation is applied.

For example, in this code, the annotation on line 9 is applied instead of the block annotation, but the block annotation is applied to the violation on line 7.

```
1 /*polyspace-begin MISRA-C:14.9 [To fix:High] "Block annotation"*/
2 int main(void) /*polyspace MISRA-C:14.7 "Nested annotation applied"*/
3 {
4     int x = 1;
5     int y = x / 2;
6
7     if (y < 0) /* Block annotation is applied to this violation of MISRA-C:14.9*/
8         y++;
9     if (x > y) /*polyspace MISRA-C:14.9 [Justified:Low] "Nested annotation applied"*/
10        return x;
11    return x;
12 }
13 /*polyspace-end MISRA-C:14.9 [To fix:High] "Block annotation"*/
```

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.

If you do not specify a status, Polyspace considers the result justified, and assigns the status **No action planned** to the result.

To replace the different annotation fields with their allowed values, use the values in this table or see the examples on page 18-9.

Field	Allowed Value
<i>Family</i>	<p>Type of analysis result:</p> <ul style="list-style-type: none"> • DEFECT (Polyspace Bug Finder) • RTE, for run-time checks (Polyspace Code Prover) • CODE-METRICS, for function-level code complexity metrics • VARIABLE, for global variables (Polyspace Code Prover) • MISRA-C or MISRA2004 for MISRA C: 2004 rule violations • MISRA-AC-AGC for violations of MISRA C:2004 rules applicable to generated code • MISRA-C3 or MISRA2012 for MISRA C: 2012 rule violations. The annotation works even for the rules applicable to generated code. • CERT-C for CERT C coding standard violations • CERT-CPP for CERT C++ coding standard violations • ISO-17961 for ISO/IEC TS 17961 coding standard violations • MISRA-CPP for MISRA C++ rule violations • AUTOSAR-CPP14 for AUTOSAR C++14 rule violations • JSF for JSF++ rule violations • CUSTOM for violations of custom coding rules <p>To specify all analysis results, use the asterisk character * : *.</p>

Field	Allowed Value
<i>Result_name</i>	<p>For DEFECT, use short names of checkers. See “Short Names of Bug Finder Defect Checkers” (Polyspace Bug Finder).</p> <p>For RTE, use short names of run-time checks. See “Short Names of Code Prover Run-Time Checks” on page 18-12.</p> <p>For CODE-METRICS, use short names of code complexity metrics. See “Short Names of Code Complexity Metrics” on page 18-14.</p> <p>For VARIABLE, the only allowed value is the asterisk character " * ".</p> <p>For coding standard violations, specify the rule number or numbers.</p> <p>To specify all parts of a result name [MISRA2012:17.*] or all result names in a family [DEFECT:*], use the asterisk character.</p>
<i>Status</i>	<p>Text to indicate how you intend to address the error in your code. This value populates the Status column in the Results List pane as:</p> <ul style="list-style-type: none"> • Unreviewed • To investigate • To fix • Justified • No action planned • Not a defect • Other <p>Polyspace suppresses results annotated with status Justified, No action planned, or Not a defect in subsequent analyses. If you specify a status that is not an allowed value, Polyspace stores it as a custom status.</p>

Field	Allowed Value
<i>Severity</i>	<p>Text to indicate how critical you consider the error in your code. This value populates the Severity column in the Results List pane as:</p> <ul style="list-style-type: none"> • Unset • High • Medium • Low <p>If you specify a severity that is not an allowed value, Polyspace appends it to the status field and stores it as a custom status. For example, [To investigate:sporadic] is displayed in the Status column of the Results List pane as To investigate sporadic.</p>
<i>Comment</i>	<p>Additional text, such as a keyword or an explanation for the status and severity. This value populates the Comment column in the Results List pane.</p> <p>The additional text can span more than one line in the code. When showing this text in reports, leading and trailing spaces on a line are merged into one space so that the entire text can be read as a single paragraph.</p>

Syntax Examples

Suppress a Single Defect

Enter an annotation on the same line as the defect and specify the *Family* (DEFECT) and the *Result_name* (INT_OVFL). When you do not specify a status, Polyspace assigns the status No action planned, and then suppresses the result in subsequent analyses.

```
int var = INT_MAX;
var++;/* polyspace DEFECT:INT_OVFL */
```

Suppress a Single Coding Standard Violation

Justify a coding standard violation, for instance, a CERT-C violation.

Enter an annotation on the same line as the violation and specify the *Family* (CERT-C) and the *Result_name* (the rule number, for instance, STR31-C). Assign the status Justified, severity Low and a comment.

```
code; /* polyspace CERT-C:STR31-C [Justified:Low] "Overflow cannot happen
because of external constraints." */
```

Suppress All MISRA C: 2012 Violations Over Multiple Lines

Enter an annotation with `+n` between `polyspace` and the `Family:Result_name` entries. The annotation applies to the same line and the `n` following lines.

This annotation applies to lines 4-7. The line count includes code, comments, and blank lines.

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

Suppress All Code Metrics on Function

To annotate function-level code complexity metrics, in the function definition, enter an annotation on the same line as the function name.

This annotation suppresses all code complexity metrics for function `func`:

```
char func(char param) { //polyspace CODE-METRICS:*
    ...
}
```

Specify Multiple Families in the Same Annotation

Enter each family separated by a space. This annotation applies to all MISRA C:2012 rules 17 and to all run-time checks.

```
some code; /* polyspace MISRA2012:17.* RTE:* */
```

Specify Multiple Result Names in the Same Annotation

After you specify the *Family* (DEFECT), enter each *Result_name* separated by a comma.

```
system("rm ~/.config"); /* polyspace DEFECT:UNSAFE_SYSTEM_CALL,RETURN_NOT_CHECKED */
```

Add Explanatory Comments to Annotation

After you specify a *Family* and a *Result_name*, you can add a *Comment* with additional information for your justification. You can provide a comment for all families and result names, or a comment for each family or result name.

```
//Single comment
code; /* polyspace DEFECT:BAD_FREE MISRA2004:* "OK Defect and MISRA" */
//Multiple comments incorrect syntax:
code; /* polyspace DEFECT:* "OK defect" MISRA2004:5.2 "OK MISRA" */
//Multiple comments correct syntax:
code; /* polyspace DEFECT:* "OK defect" polyspace MISRA2004:5.2 "OK MISRA" */
```

In annotations, Polyspace ignores all text following double quotes. To specify additional *Family:Result_name*, [*Status:Severity*] or *Comment* entries, you must reenter the keyword `polyspace` after text in double quotes.

Set Status and Severity

You can specify allowed values on page 18-6 or enter custom values for status and severity. A custom severity entry is appended to the status and stored as a custom **Status** in the user interface.

```
//Set Status only
code; /* polyspace DEFECT:* [To fix] "some comment" */

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

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

See Also

-xml-annotations-description

More About

- “Define Custom Annotation Format” on page 18-20
- “Short Names of Code Prover Run-Time Checks” on page 18-12
- “Short Names of Code Complexity Metrics” on page 18-14

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	ABS_ADDR
AUTOSAR runnable not implemented	AUTOSAR_NOIMPL
Correctness condition	COR
Division by zero	ZDV
Function not called	FNC
Function not reachable	FNR
Function returns a value	FRV
Illegally dereferenced pointer	IDP
Incorrect object oriented programming	OOP
Invalid C++ specific operations	CPP
Invalid floating point operation	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” on page 18-6

Short Names of Code Complexity Metrics

When annotating your code to justify metrics or creating custom software quality objectives, you use short names of code complexity metrics instead of the full names. The following table lists the short names for code complexity metrics.

Note that you can only annotate your code for function level code complexity metrics only.

Project Metrics

Code Metric	Acronym
Number of Direct Recursions	AP_CG_DIRECT_CYCLE
Number of Header Files	INCLUDES
Number of Files	FILES
Number of Protected Shared Variables (Code Prover only)	PSHV
Number of Recursions	AP_CG_CYCLE
Number of Potentially Unprotected Shared Variables (Code Prover only)	UNPSHV
Program Maximum Stack Usage (Code Prover only)	PROG_MAX_STACK
Program Minimum Stack Usage (Code Prover only)	PROG_MIN_STACK

File Metrics

Code Metric	Acronym
Comment Density	COMF
Estimated Function Coupling	FCO
Number of Lines	TOTAL_LINES
Number of Lines Without Comment	LINES_WITHOUT_CMT

Function Metrics

Code Metric	Acronym
Cyclomatic Complexity	VG
Higher Estimate of Local Variable Size	LOCAL_VARS_MAX
Language Scope	VOCF
Lower Estimate of Local Variable Size	LOCAL_VARS_MIN
Minimum Stack Usage (Code Prover only)	MIN_STACK
Maximum Stack Usage (Code Prover only)	MAX_STACK
Number of Call Levels	LEVEL

Code Metric	Acronym
Number of Call Occurrences	NCALLS
Number of Called Functions	CALLS
Number of Calling Functions	CALLING
Number of Executable Lines	FXLN
Number of Function Parameters	PARAM
Number of Goto Statements	GOTO
Number of Instructions	STMT
Number of Lines Within Body	FLIN
Number of Local Non-Static Variables	LOCAL_VARS
Number of Local Static Variables	LOCAL_STATIC_VARS
Number of Paths	PATH
Number of Return Statements	RETURN

See Also

More About

- “Annotate Code and Hide Known or Acceptable Results” on page 18-6

Annotate Code for Known or Acceptable Results (Not Recommended)

Note Starting R2017b, Polyspace uses a simpler annotation format. See “Annotate Code and Hide Known or Acceptable Results” on page 18-6.

If Polyspace finds defects in your code that you cannot or will not fix, you can add annotations to your code. These annotations are code comments that indicate known or acceptable defects or coding rule violations. By using these annotations, you can:

- Avoid rereviewing defects or coding rule violations from previous analyses.
- Preserve review comments and classifications.

Note Source code annotations do not apply to code comments. You cannot annotate these rules:

- MISRA C:2004 Rules 2.2 and 2.3
 - MISRA C:2012 Rules 3.1 and 3.2
 - MISRA-C++ Rule 2-7-1
 - JSF++ Rules 127 and 133
-

Add Annotations from the Polyspace Interface

This method shows you how to convert review comments and classifications in the Polyspace interface into code annotations.

- 1 On the **Results List** or **Result Details** pane, assign a **Severity**, **Status**, and **Comment** to a result.
 - a Click a result.
 - b From the **Severity** and **Status** dropdown lists, select an option.
 - c In the **Comment** field, enter a comment or keyword that helps you easily recognize the result.
- 2 On the **Results List** pane, right-click the commented result and select **Add Pre-Justification to Clipboard**. The software copies the severity, status, and comment to the clipboard.
- 3 Right-click the result again and select **Open Editor**. The software opens the source file to the location of the defect.
- 4 Paste the contents of your clipboard on the line immediately before the line containing the defect or coding rule violation.

You can see your review comments as a code comment in the Polyspace annotation syntax, which Polyspace uses to repopulate review comments on your next analysis.

- 5 Save your source file and rerun the analysis.

On the **Results List** pane, the software populates the **Severity**, **Status**, and **Comment** columns for the defect or rule violation that you annotated. These fields are read only because they are

populated from your code annotation. If you use a specific keyword or status for your annotations, you can filter your results to hide or show your annotated results. For more information on filtering, see “Filter and Group Results” on page 19-2.

Add Annotations Manually

This method shows you how to enter comments directly into your source files by using the Polyspace code annotation syntax. The syntax is not case-sensitive and applies to the first uncommented line of C/C++ code following the annotation.

- 1 Open your source file in an editor and locate the line or section of code that you want to annotate.
- 2 Add one of the following annotations:
 - For a single line of code, add the following text directly before the line of code that you want to annotate.

```
/* polyspace<Type:Kind1[,Kind2] : [Severity] : [Status] > [Notes] */
```

- For a section of code, use the following syntax.

```
/* polyspace:begin<Type:Kind1[,Kind2] : [Severity] : [Status] > [Notes] */
```

```
... Code section ...
```

```
/* polyspace:end<Type:Kind1[,Kind2] : [Severity] : [Status] > */
```

If a macro expands to multiple lines, use the syntax for code sections even though the macro itself covers one line. The single-line syntax applies only to results that appear in the first line of the expanded macro.

- 3 Replace the words *Type*, *Kind1*, [*Kind2*], [*Severity*], [*Status*], and [*Additional text*] with allowed values, indicated in the following table. The text with square brackets [] is optional and you can delete it. See “Syntax Examples” on page 18-18.

Word	Allowed Values
<i>Type</i>	<p>The type of results:</p> <ul style="list-style-type: none"> • Defect (Polyspace Bug Finder) • RTE, for run-time checks (Polyspace Code Prover) • VARIABLE, for global variables (Polyspace Code Prover) • CODE-METRIC, for code complexity metrics. • MISRA-C, for MISRA C:2004 • MISRA-AC-AGC • MISRA-C3, for MISRA C:2012 • MISRA-CPP • JSF • Custom, for custom coding rule violations.

Word	Allowed Values
<i>Kind1, [Kind2], ...</i>	<p>For defects, run-time checks and code metrics, use the short names of checkers. See:</p> <ul style="list-style-type: none"> • “Short Names of Bug Finder Defect Checkers” (Polyspace Bug Finder) • “Short Names of Code Prover Run-Time Checks” on page 18-12 <p>For coding rule violations, specify the rule number or numbers.</p> <p>For global variables, the only allowed value is ALL.</p> <p>If you want the comment to apply to all possible defects or coding rules, specify ALL.</p>
<i>Severity</i>	<p>Text that indicates how critical you consider the defect. Enter one of the following:</p> <ul style="list-style-type: none"> • Unset • High • Medium • Low <p>This text populates the Severity column on the Results List pane.</p>
<i>Status</i>	<p>Text that indicates how you intend to correct the error in your code. Enter one of the following or any other text:</p> <ul style="list-style-type: none"> • Unreviewed • To investigate • To fix • Justified • No action planned • Not a defect • Other <p>This text populates the Status column on the Results List pane. The status is also used in Polyspace Metrics to determine whether a result is justified. To justify a result, use Justified, No action planned or Not a defect.</p>
<i>Notes</i>	<p>Additional comments, such as a keyword or an explanation for the status and severity.</p>

Syntax Examples

- A single defect:

```
/* polyspace<Defect:HARD_CODED_BUFFER_SIZE:Medium:To investigate> Known issue */
int table[100];
```

- A single run-time check:

```
/* polyspace<RTE: ZDV : High : To Fix > Denominator cannot be zero */
y=1/x;
```

- A MISRA C:2012 rule violation:

```
/* polyspace<MISRA-C3: 13.1 : Low : Justified> Known issue */  
int arr[2] = {x++,y};
```

- Unused global variable:

```
/* polyspace<VARIABLE: ALL : Low : Justified> Variable to use later*/  
int var_unused;
```

- Multiple defects:

```
polyspace<Defect:USELESS_WRITE,DEAD_CODE:Low:No Action Planned> OK issue
```

- Multiple JSF rule violations:

```
polyspace<JSF:9,13:Low:Justified> Known issue
```

Define Custom Annotation Format

This example shows how to create and edit an XML file to define an annotation format and map it to the Polyspace annotation syntax.

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*)*([\^-\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” (MATLAB). It is recommended that you include a keyword in the pattern of your annotation syntax to help identify it. In this example, the keyword is `myKeyword`. Set the attribute `Search_For_Keywords` equal to this keyword.

Once you know the pattern of your annotation, you can define it in the XML by adding an `<Expression/>` element and specifying at least the attributes `Mode`, `Regex`, and `Rule_Identifier_Position`. For instance, the first `<Expression/>` element in `annotations_description.xml` defines an annotation with these attributes:

- `Mode="SAME_LINE"`. The annotation applies to code on the same line.
- `Regex="myKeyword\s+(\w+(\s*,\s*\w+)*)"`. Polyspace uses the regular expression to search for a string that begins with `myKeyword`, followed by a space `\s+`. Polyspace then searches for a capturing group `(\w+(\s*,\s*\w+)*)` that includes an alphanumeric rule identifier `\w+` and, optionally, additional comma-separated rule identifiers `(\s*,\s*\w+)*`.
- `Rule_Identifier_Position="1"`. The integer value of this attribute corresponds to the number of opening parentheses preceding the relevant capturing group in the regular expression. In `myKeyword\s+(\w+(\s*,\s*\w+)*)`, one opening parenthesis precedes the capturing group of the rule identifier `(\w+(\s*,\s*\w+)*)`. If you want to match rule identifiers captured by `(\s*,\s*\w+)`, then you set `Rule_Identifier_Position="2"` because two opening parentheses precede this capturing group.

The list of attributes and their values are listed in this table. The example column refers to the format defined in `annotations_description.xml`.

Attribute	Use	Value	Example
Mode	Required	SAME_LINE	Applies only on the same line as the annotation. code; //myKeyword 100
		GOTO_INCREMENT	Applies on the same line as the annotation and the following n lines: 3. code; // myKeyword +3 ALL_MISRA 4. /*comments */ 5. 6. code; 7. code; The preceding annotation applies to lines 3-6 only.

Attribute	Use	Value	Example
		BEGIN	<p>Applies to the same line and all following lines until a corresponding expression with attribute Mode="END" or "END_ALL", or until the end of the file.</p> <pre>//myKeyword 50, 51 Block_on Code block 1; ...</pre>
		END	<p>Stops the application of a rule identifier declared by a corresponding expression with attribute Mode="BEGIN".</p> <pre>//myKeyword 50, 51 Block_on Code block 1; ... More code; //myKeyword 50 Block_off</pre> <p>Only rule identifier 50 is turned off. Rule identifier 51 still applies.</p>
		END_ALL	<p>Stops all rule identifiers declared by an expression with attribute Mode="BEGIN".</p> <pre>//myKeyword 50, 51 Block_on Code block 1; ... More code; //myKeyword Block_off_all</pre> <p>Rule identifiers 50 and 51 are turned off.</p>
Regex	Required	Regular expression search string	<p>See "Regular Expressions" (MATLAB). Regex="myKeyword\s+(\w+(\s*,\s*\w+)*)" matches these expressions:</p> <pre>// myKeyword 50, 51 /* myKeyword ALL_MISRA, 100 */</pre>

Attribute	Use	Value	Example
Rule_Identifier_Position	Required, except when you set Mode="END_ALL"	Integer	<p>The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.</p> <pre><Expression Mode="GOTO_INCREMENT" Regex="myKeyword\s+(\+\d+\s) (\w+(\s*,\s*\w+))*" Increment_Position="1" Rule_Identifier_Position="2"/></pre> <p>Note Enter the regex expression on a single line when you edit your XML file.</p> <p>The search expression for the rule identifier <code>\w+(\s*,\s*\w+)*</code> is after the second opening parenthesis of the regular expression.</p>
Increment_Position	Required only when you set Mode="GOTO_INCREMENT"	Integer	<p>The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.</p> <pre><Expression Mode="GOTO_INCREMENT" Regex="myKeyword\s+(\+\d+\s) (\w+(\s*,\s*\w+))*" Increment_Position="1" Rule_Identifier_Position="2"/></pre> <p>Note Enter the regex expression on a single line when you edit your XML file.</p> <p>The search expression for the increment <code>\+\d+\s</code> is after the first opening parenthesis of the regular expression.</p>
Status_Position	Optional	Integer	<p>See Increment_Position example. When you use this attribute, the entry in your annotation is displayed in the Status column on the Results List pane of the user interface.</p>
Severity_Position	Optional	Integer	<p>See Increment_Position example. When you use this attribute, the entry in your annotation is displayed in the Severity column on the Results List pane of the user interface.</p>

Attribute	Use	Value	Example
Comment_Position	Optional	Integer	See Increment_Position example. When you use this attribute, the entry in your annotation is displayed in the Comment column on the Results List pane of the user interface. Your comment is appended to the string 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	See the mapping section of <code>annotations_description.xml</code>
Family	Required	Corresponds to Polyspace results family. For a list of allowed values, see allowed values on page 18-12.	See the mapping section of <code>annotations_description.xml</code>
Result_Name	Required	Corresponds to Polyspace result names. For a list of allowed values, see allowed values on page 18-12.	See the mapping section of <code>annotations_description.xml</code>

See Also

"Annotation Description Full XML Template" on page 18-27 | `-xml-annotations-description`

More About

- “Annotate Code and Hide Known or Acceptable Results” on page 18-6

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 18-20.

Element	Attribute	Use	Value
Annotations	Group	Required	User defined string. For example, "Custom Annotations"
Expressions	Search_For_Keyword s	Required	User defined string. This string is a keyword you include in the pattern of your annotation syntax to help identify it. For example, "myKeyword"
	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
			END_ALL

Element	Attribute	Use	Value
			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 18-12.
	Result_Name	Required	Corresponds to Polyspace result names. For a list of allowed values, see allowed values on page 18-12.

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 18-6

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.

Select one or more results to review:

- ✓ **Overflow**
- ▼ MISRA C:2012 10.3 (Required)
- ▼ MISRA C:2012 10.3 (Required) ?

The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category. The expression is assigned to an object with a different essential type category.

Source

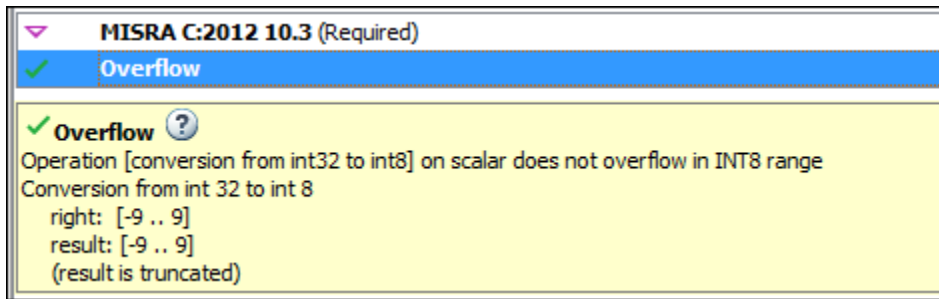
file2.c x

```

1  int func (int var1, int var2) {
2      char temp;
3      temp = var1 - var2;
4      if (temp > 0)
5          return -1;
6      else
7          return 1;
8  }

```

The green check indicates that the conversion from `int` to `char` does not overflow.



You can use the green overflow check as rationale to justify the rule violation.

Rules About Pointer Arithmetic

Pointer arithmetic on nonarray pointers are okay if the pointers stay within the allowed buffer.

In the following example, the operation `ptr++` violates MISRA C:2004 Rule 17.4. The rule states that array indexing shall be the only allowed form of pointer arithmetic. Here, a pointer that is not an array is incremented.

```

#define NUM_RECORDS 3
#define NUM_CHARACTERS 6

void readchar(char);

int main(int argc, char* argv[]) {
    char dbase[NUM_RECORDS][NUM_CHARACTERS] = { "r5cvx", "a2x5c", "g4x3c" };
    char *ptr = &dbase[0][0];
    for (int index = 0; index < NUM_RECORDS * NUM_CHARACTERS; index++) {
        readchar(*ptr);
        ptr++;
    }
    return 0;
}

```

Consider the rationale behind this rule. After an increment, a pointer can go outside the bounds of an allowed buffer (such as an array) or even point to an arbitrary location. Pointer arithmetic is fine as long as the pointer points within an allowed buffer. You can justify this particular rule violation by using the results of a Code Prover `Illegally dereferenced pointer` check.

Run Code Prover on this code. On the **Source** pane, click the `++` in `ptr++`. You see the expected violation of MISRA C:2004 Rule 17.4.

Result Review

Severity Enter comment here...
Status

MISRA C:2004 17.4 (Required) ?
Array indexing shall be the only allowed form of pointer arithmetic.

Configuration Result Details

Source

file3.c X

```

1  #define NUM_RECORDS 3
2  #define NUM_CHARACTERS 6
3
4  void readchar(char);
5
6  int main(int argc, char* argv[]) {
7      char dbase[NUM_RECORDS][NUM_CHARACTERS] = { "r5cvx", "a2x5c", "g4x3c"};
8      char *ptr = &dbase[0][0];
9      for (int index = 0; index < NUM_RECORDS * NUM_CHARACTERS; index++) {
10         readchar(*ptr);
11         ptr++;
12     }
13     return 0;
14 }

```

Click the * on the operation `readchar(*ptr)`. You see a green **Illegally dereferenced pointer** check. The green check indicates that the pointer points within allowed bounds when dereferenced.

✓ Illegally dereferenced pointer ?

Pointer is within its bounds
Dereference of local pointer 'ptr' (pointer to int 8, size: 8 bits):
 Pointer is not null.
 Points to 1 bytes at offset [0 .. 17] in buffer of 18 bytes, so is within bounds (if memory is allocated).
 Pointer may point to variable or field of variable:
 'dbase', local to function 'main'.

You can use the green check to justify the rule violation.

See Also

Related Examples

- “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2

Manage Results

- “Filter and Group Results” on page 19-2
- “Prioritize Check Review” on page 19-9

Filter and Group Results

When you open the results of a Polyspace analysis, you see a flat list of defects (Bug Finder), run-time checks (Code Prover), coding rule violations or other results. To organize your review, you can narrow down the list or group results by file or result type.

The diagram illustrates the 'Results List' window and how it can be filtered and grouped. The top window shows a flat list of 397 results. Two arrows point to the bottom windows: 'Filter results' and 'Group results'.

Filter results: The 'Results List' window shows a filtered list of 5 results. The 'Showing' indicator displays 'Showing 5/397'.

Family	Group	Check
● *	Static memory	Out of bounds array index
● *	Static memory	Illegally dereferenced pointer
● *	Other	Invalid use of standard library routine
● *	Control flow	Non-terminating call
● *	Control flow	Non-terminating loop

Group results: The 'Results List' window shows results grouped by file (example.c) and function (Close_To_Zero(), File Scope, get_oil_pressure(), Non_Infinite_Loop(), Pointer_Arithmetic(), Illegally dereferenced pointer).

Family	Check
example.c	Close_To_Zero()
example.c	File Scope
example.c	get_oil_pressure()
example.c	Non_Infinite_Loop()
example.c	Pointer_Arithmetic()
example.c	Illegally dereferenced pointer

Some of the ways you can use filtering are:

- You can display certain types of defects or run-time checks only.

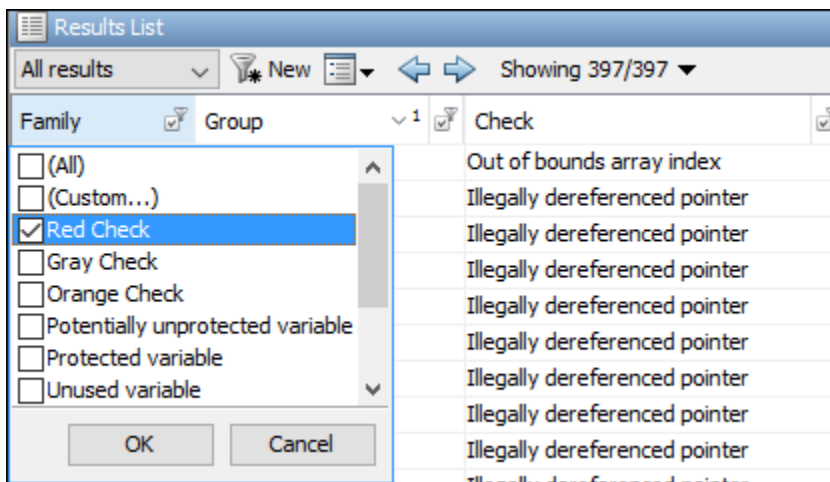
For instance, in Bug Finder, you can display only high-impact defects. See “Classification of Defects by Impact” (Polyspace Bug Finder).

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

For information on justification, see “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

Filter Results

Filter Using Results List



You can filter using the columns on the **Results List** pane. Click the  icon on the column headers to see the available filters. For information on the columns, see:

- “Results List” (Polyspace Bug Finder)
- “Results List” on page 16-21

Results found since the last analysis appear with an asterisk (*) next to them. To see only these results, click the **New** button.

If you do not want to filter by the exact contents of a column, you can use a custom filter instead. For instance, you want to filter out subfolders of a specific folder. Instead of filtering out each subfolder in the **Folder** column, select **Custom** from the filter dropdown. Specify the root folder name for the doesn't contain filter.

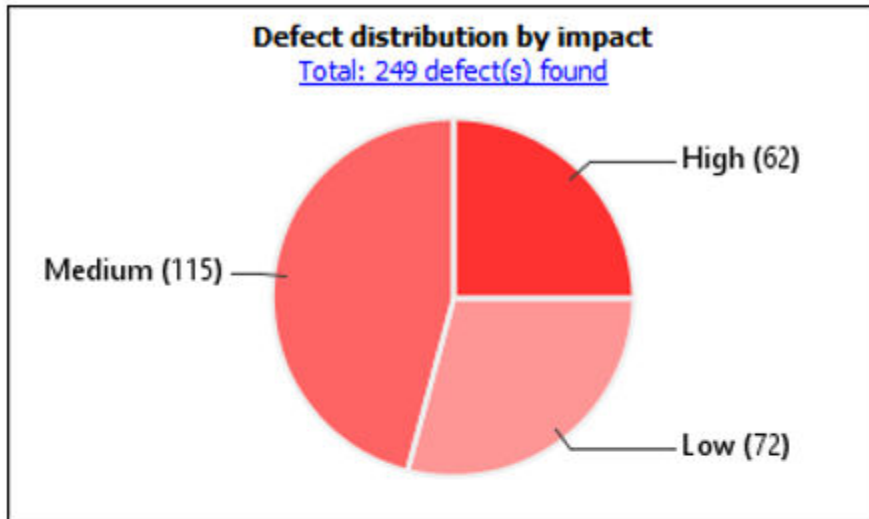
You can use wildcard characters for the custom filter. The wildcard ? represents 0 or 1 character and * represents 0 or more characters.

If you apply filters in this way, they carry over to the next analysis. You can also name and save a subset of filters for use in multiple projects. To apply the named set of filters, pick this filter set from

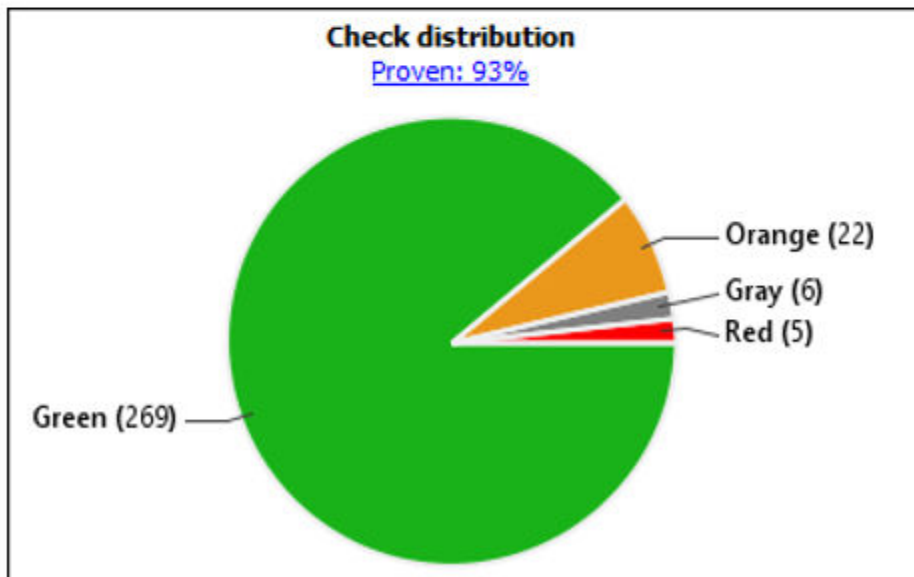
the **All results** list. To create a new entry in this list, select **Tools > Preferences** and create your own set of filters on the **Review Scope** tab.

Filter Using Dashboard

Bug Finder



Code Prover



You can click graphs on the **Dashboard** pane to filter results. For instance:

- To see only high-impact defects in Bug Finder, click the corresponding section of the **Defect distribution by impact** chart.
- To see only red checks in Code Prover, click the corresponding section of the **Check distribution** chart.

To see all results again, click the link **View all results in this scope**.

Filter Using Orange Sources

An orange source can cause multiple orange checks in Code Prover. You can display all orange checks from the same source and review them together.

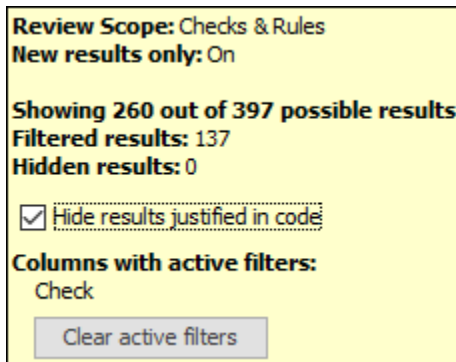
For instance, in this code, the unknown value `input` can cause an overflow and a division by zero. The variable `input` is an orange source that causes two orange checks.

```
void func (int input) {
int val1;
double val2;
val1 = input++;
val2 = 1.0/input;
}
```

To begin, select **Window > Show/Hide View > Orange Sources**. You see the list of orange sources. Select an orange source to see all orange checks coming from this source.

Orange Sources				
Source Type	Name	File	Line	Max Oranges
stubbed function	get_bus_status()		-1	1
stubbed function	random_float()		-1	3
stubbed function	random_int()		-1	1
local volatile variable	get_oil_pressure.vol_j	example.c	27	2
local volatile variable	all_values_s32.tmps32	single_file_analysis.c	29	2

See Filters Used



On the **Results List** header, you see the number of results displayed in the format **Showing x/y**, for instance **Showing 100/250**. Click the dropdown beside this number to see the filters that are currently active. You can also clear the active filters from this dropdown (all except the named set of filters that you picked from the **All results** dropdown).


You see this information about the filters:

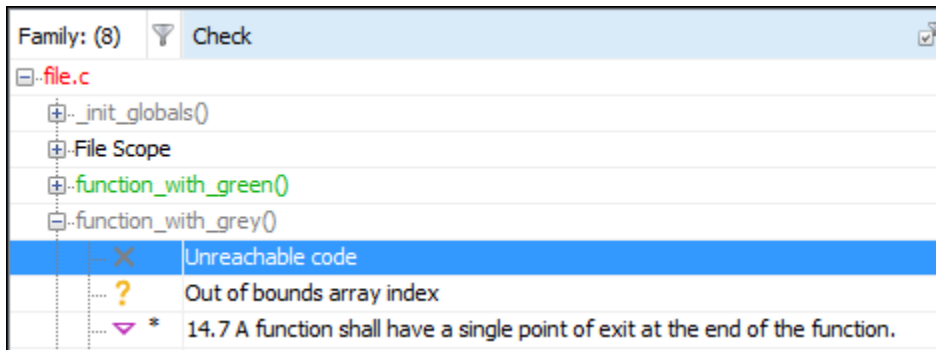
- **Review Scope:** If you pick a named set of filters from the **All results** dropdown, you see this filter set.
- **New results only:** If you use the **New** button to see only new results, you see this filter enabled.
- **Filtered results:** You see the number of results filtered in the Polyspace user interface (by any means: results list, dashboard or orange sources).
- **Hidden results:** You see the number of results hidden using code annotations. To unhide these results, clear **Hide results justified in code**.

For information on hiding results through code annotations, see “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

- **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” (Polyspace Bug Finder).
- 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 19-9

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 22-72. 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 Polyspace Code Prover Results” on page 16-2.

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	<ul style="list-style-type: none"> • Out of bounds array index • Non-initialized local variable • Division by zero • Invalid shift operations


Review Order	Checks
Second	<ul style="list-style-type: none"> • 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” on page 19-6.

- 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” on page 16-51.

- 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” on page 19-2

Generate Reports from Polyspace Results

- “Generate Reports” on page 20-2
- “Export Polyspace Analysis Results” on page 20-5
- “Export Polyspace Analysis Results to Excel by Using MATLAB Scripts” on page 20-7
- “Export Global Variable List” on page 20-9
- “Visualize Code Prover Analysis Results in MATLAB” on page 20-13
- “Customize Existing Code Prover Report Template” on page 20-16
- “Sample Report Template Customizations” on page 20-21

Generate Reports

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

To generate reports, you can do one of the following:

- Run a Polyspace analysis and create a report from the analysis results. See the workflow described here.
- Specify that a report will be automatically generated after analysis. For more information on the options, see “Reporting”.
- Export your results to a text file and generate graphs and statistics. See “Export Polyspace Analysis Results” on page 20-5.

Depending on the template you use, the report contains information about certain types of results from the **Results List** pane. You can see the following information about a result:

- ID: Unique number for a result for the current analysis

To identify the result in your source code, you can use the ID in the **Results List** pane of the Polyspace user interface or in your IDE if you are using a Polyspace plugin.

- Check: Defect names, MISRA C:2012 coding rule number, and so on.
- File and function
- Status, Severity, Comment: Information that you enter about a result.

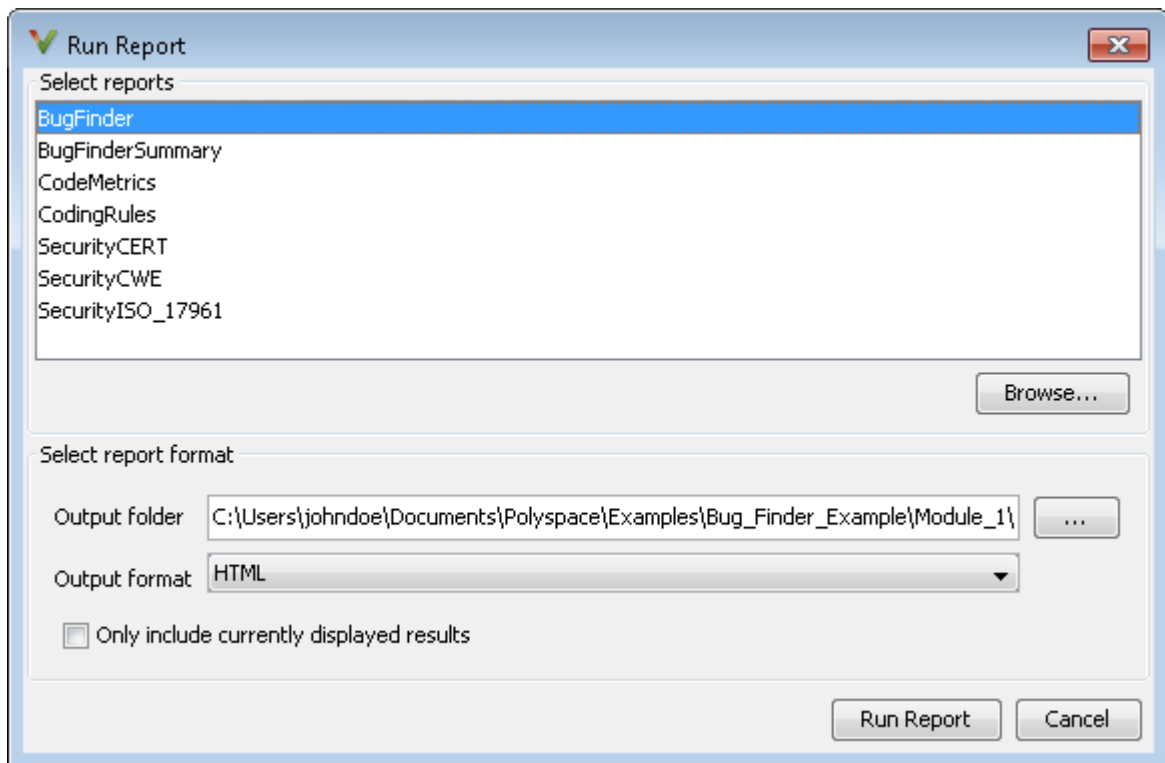
In Bug Finder, the report does not contain the line or column number for a result. Use the report for archiving, gathering statistics and checking whether results have been reviewed and addressed (for certification purposes or otherwise). To review a result in your source code, use the Polyspace user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

Generate Reports from User Interface

You can generate a report from your analysis results. Using a customizable template, the report presents your results in a concise manner for managerial review or other purposes.

- 1 Open your results file.
- 2 Select **Reporting > Run Report**.

The Run Report dialog box opens.



3 Select the following options:

- In the **Select Reports** section, select the types of reports that you want to generate. Press the **Ctrl** key to select multiple types. For example, you can select **BugFinder** and **CodeMetrics**.
- Select the **Output folder** in which to save the report.
- Select an **Output format** for the report.
- If the display language (Windows) or locale (Linux) of your operating system is set to another language, you see an option to generate English reports. Select this option if you want an English report, otherwise the report is in another language.
- If you want to filter results from your report, use filters on the **Results List** pane to display only the results that you want to report. Then, when generating reports, select **Only include currently displayed results**. You cannot display filtered reports for results downloaded from Polyspace Metrics.

For more information on filtering, see “Filter and Group Results” on page 19-2.

4 Click **Run Report**.

The software creates the specified report and opens it.

Generate Reports from Command Line

You can script the generation of reports using the `polyspace-report-generator` command.

To generate **BugFinder** and **CodeMetrics** HTML reports for results in `C:\Users\johndoe\Documents\Polyspace\Examples\Bug_Finder_Example\Module_1\BF_Result`, use the following options with the command:

```
SET template_path=^
"C:\Program Files\MATLAB\R2018a\toolbox\polyspace\psrptgen\templates\bug_finder"
SET bf_templates=^
%template_path%\BugFinder.rpt,%template_path%\CodingMetrics.rpt
SET results_dir=^
"C:\Users\johndoe\Documents\Polyspace\Examples\Bug_Finder_Example\Module_1\BF_Result"

polyspace-report-generator ^
-results-dir %results_dir% ^
-template %bf_templates ^
-format html
```

See Also

Generate report | Bug Finder and Code Prover report (-report-template) | Output format (-report-output-format)

More About

- “Customize Existing Code Prover Report Template” on page 20-16
- “Export Polyspace Analysis Results” on page 20-5

Export Polyspace Analysis Results

You can export your analysis results to a tab delimited text file or a MATLAB table (MATLAB). Using the text file or table, you can:

- Generate graphs or statistics about your results that you cannot readily obtain from the user interface by using MATLAB or Microsoft Excel®. For instance, for each Code Prover check type (**Division by zero**, **Overflow**), you can calculate how many checks are red, orange, or green.
- Integrate the analysis results with other checks you perform on your code.

Export Results to Text File

You can export results from the user interface or command line.

User Interface	Command Line
<p>1 Open your analysis results.</p> <p>2 Export all results or only a subset of the results.</p> <ul style="list-style-type: none"> • To export all results, select Reporting > Export > Export All Results. • If you want to filter results from your report, use filters on the Results List pane to display only the results that you want to report. Then, when exporting results, select Reporting > Export > Export Currently Displayed Results. <p>For more information on filtering, see “Filter and Group Results” on page 19-2.</p> <p>3 Select a location to save the text file and click OK.</p>	<p>Use appropriate options with the <code>polyspace-report-generator</code> command.</p> <p>The available options are:</p> <ul style="list-style-type: none"> • <code>-generate-results-list-file</code>: Specifies that a text file must be generated. This option is required. • <code>-results-dir <i>folder_paths</i></code>: Path to folder containing your analysis results. If you do not specify a folder path, the software uses analysis results from the current folder. <p>To generate text files for multiple analyses, specify <code>folder_paths</code> as comma separated list with no spaces after the commas. For example:</p> <pre>C:\My_project \Module_1\results,C:\My_project \Module_2\Results</pre> <p>To merge the text files, use the <code>join</code> function.</p> <ul style="list-style-type: none"> • <code>-set-language-english</code>: 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.

The exported text file uses the character encoding on your operating system. If special characters from your comments are not exported correctly in the text file, change the character encoding on your operating system before exporting.

Export Results to MATLAB Table

Instead of a text file, you can read your Polyspace analysis results into a MATLAB table. See:

- “Visualize Bug Finder Analysis Results in MATLAB” (Polyspace Bug Finder)
- “Visualize Code Prover Analysis Results in MATLAB” on page 20-13

View Exported Results

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

- “Results List” (Polyspace Bug Finder)
- “Results List” on page 16-21

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

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

When you merge analysis results from multiple modules that contain common files, use this entry to eliminate duplicates. For instance, if you run coding-rule checking on two different modules and merge the results, coding rule violations in common header files appear twice in the results. To eliminate duplicates, compare containing files and keys of results. If two results have the same files and keys, one is a duplicate of the other.

You cannot identify the location of a Bug Finder result in your source code via the text file. However, you can still parse the file and generate graphs or statistics about your results.

See Also

Related Examples

- “Visualize Code Prover Analysis Results in MATLAB” on page 20-13
- “Export Global Variable List” on page 20-9
- “Export Polyspace Analysis Results to Excel by Using MATLAB Scripts” on page 20-7

Export Polyspace Analysis Results to Excel by Using MATLAB Scripts

You can export the results of a Bug Finder or Code Prover analysis to an Excel report. See “Export Polyspace Analysis Results” on page 20-5. 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 (MATLAB).
- `writetable`: Write a MATLAB table to a file. If the file name has the extension `.xlsx`, the function writes to an Excel file.

To specify the content to write to the Excel sheet, use these name-value pairs:

- Use the name `Sheet` paired with a sheet name to specify a worksheet in the Excel workbook.
- Use the name `Range` paired with a cell name to specify the starting cell in the worksheet where the writing begins.

Control Formatting of Excel Report

Though you can control the content exported to the Excel report by using the preceding method, the method has limited formatting options for the report.

To format the Excel report on Windows systems, access the COM server directly by using `actxserver`. For example, Technical Solution 1-QLD4K uses `actxserver` to establish a connection between MATLAB® and Excel, write data to a worksheet, and specify the colors of the cells.

See also “Get Started with COM” (MATLAB).

See Also

More About

- “Export Polyspace Analysis Results” on page 20-5

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 (MATLAB). 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” on page 10-2.

Export Variable List to Text File

You can export results from the user interface or command line.

User Interface	Command Line
<ol style="list-style-type: none"> 1 Open your verification results. 2 Select Reporting > Export > Export Variable Access. 3 Select a location to save the text file and click OK. 	<p>Use appropriate options with the <code>polyspace-report-generator</code> command.</p> <p>The available options are:</p> <ul style="list-style-type: none"> • <code>-generate-variable-access-file</code>: Specifies that a text file must be generated. • <code>-results-dir <i>folder_paths</i></code>: Path to folder containing your analysis results. If you do not specify a folder path, the software uses analysis results from the current folder. <p>To generate text files for multiple analyses, specify <code>folder_paths</code> as comma separated list with no spaces after the commas. For example:</p> <pre>C:\My_project \Module_1\results,C:\My_project \Module_2\Results</pre> <p>To merge the text files, use the <code>join</code> function.</p> <ul style="list-style-type: none"> • <code>-set-language-english</code>: 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.

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” on page 16-36.

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” on page 20-5
- “Variable Access” on page 16-36

Visualize Code Prover Analysis Results in MATLAB

After analysis, you can read your results to a MATLAB table (MATLAB). 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” on page 3-5.

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);

```

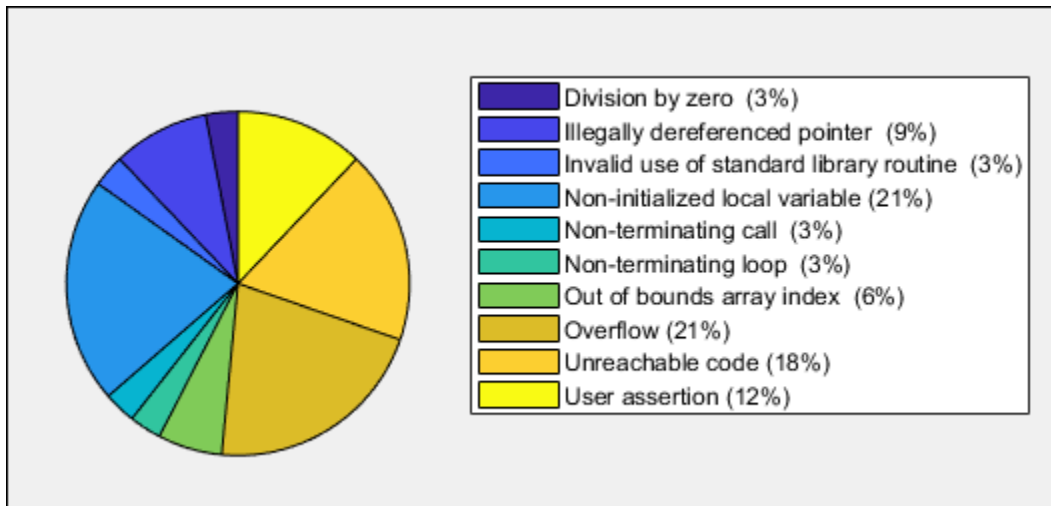
The key functions used in the example are:

- `polyspace.CodeProverResults`: Read Code Prover results into table (MATLAB).
- `pie`: Create pie chart from a categorical array (MATLAB). 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” on page 20-5
- “Export Polyspace Analysis Results to Excel by Using MATLAB Scripts” on page 20-7

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” on page 20-2.

- 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” on page 3-2.

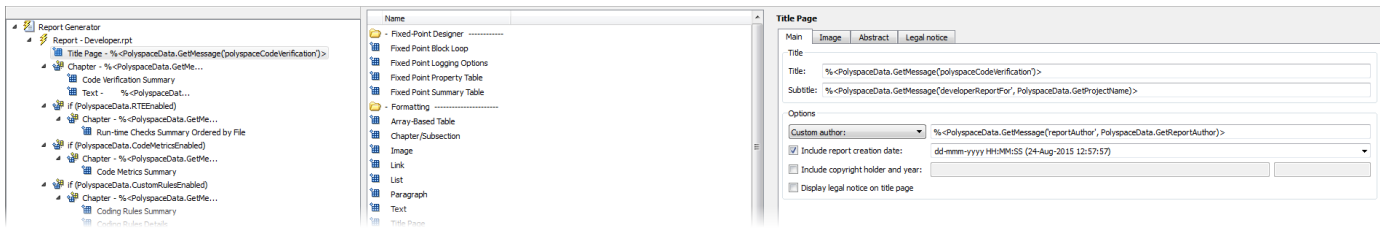
- 2 Open the Report Explorer interface of Simulink Report Generator. At the MATLAB command prompt, enter:

```
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 the MATLAB Report Generator documentation. 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:

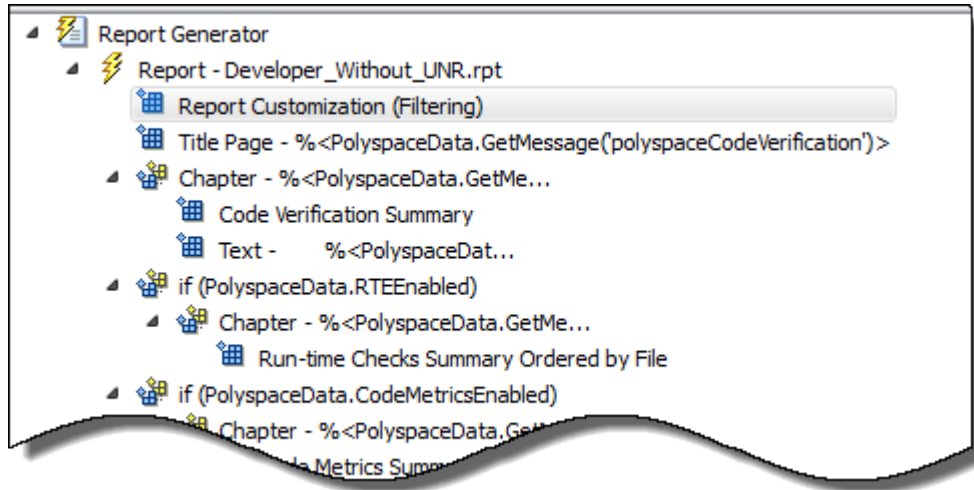
- Change properties of existing components of your template.
- Add new components to your template or remove existing components.

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.

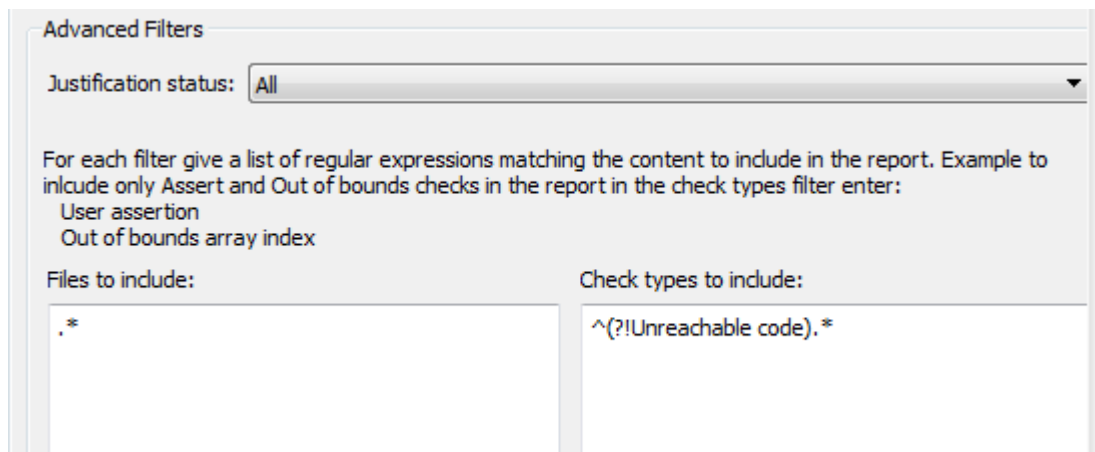
To perform this action:

- a Drag the component **Report Customization (Filtering)** from the middle pane and drop it above the **Title Page** component. The positioning of the component ensures that the filters apply to the full report and not one chapter of the report.



- b Select the **Report Customization (Filtering)** component. On the right pane, you can set the properties of this component. By default, the properties are set such that all results are included in the report.

To 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” (MATLAB) and “Polyspace Code Prover Results”.

You can toggle between activating and deactivating this component. Right-click the component and select **Activate/Deactivate Component**.

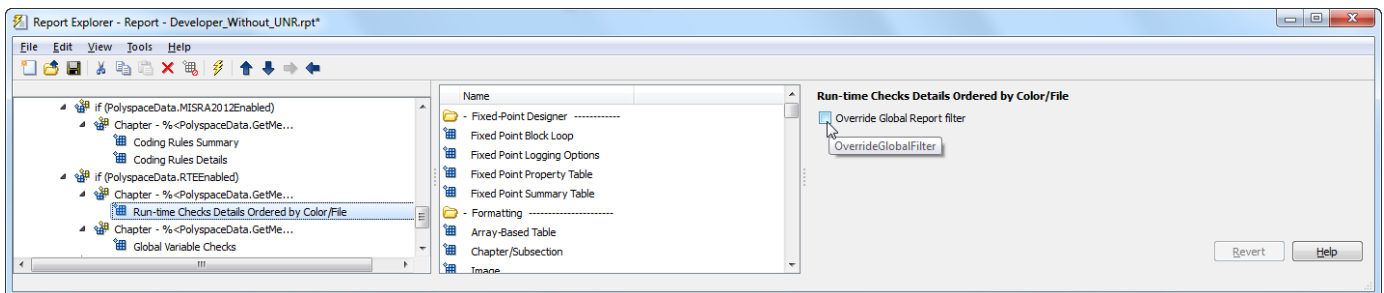
- 3 Change an existing chapter-specific component so that it does not override the global filter you applied in the previous step. If you prevent the overriding, the chapter-specific component follows the filtering specifications in the global component.

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.



- 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.psrj**.

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**. 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**. 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 20-21.

See Also

Bug Finder and Code Prover report (-report-template) | Generate report | Output format (-report-output-format)

Related Examples

- “Generate Reports” on page 20-2
- “Sample Report Template Customizations” on page 20-21

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 20-16.

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.

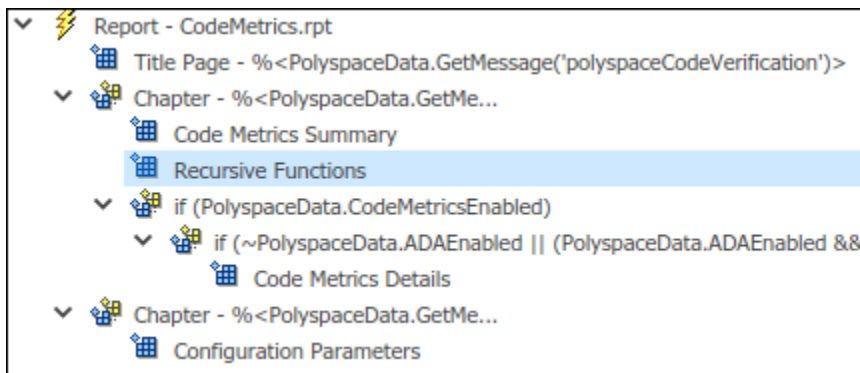
- 3 Add, remove, or modify components of the template.

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.

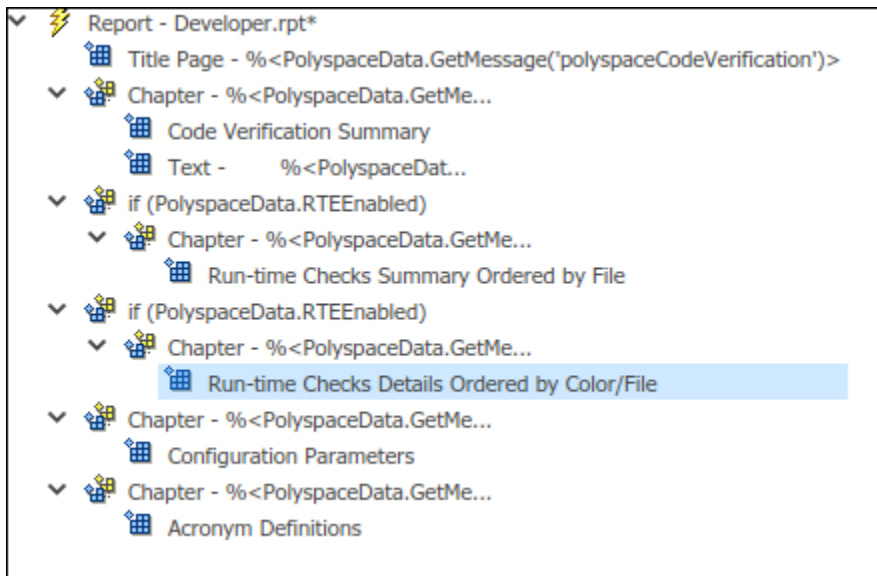
Start from the **Developer** template. Remove all chapters, except the ones containing these components:

- 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.



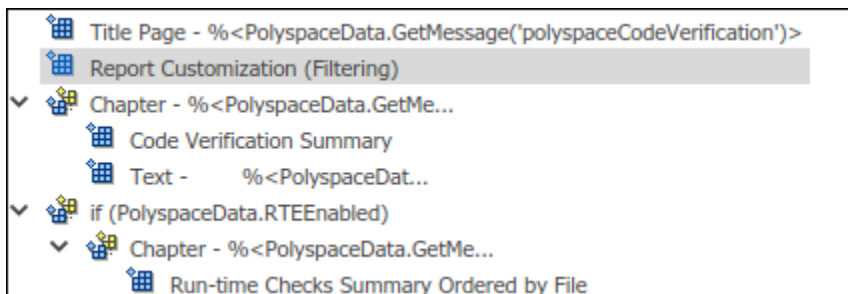
When you generate a report by using the modified template, you see an overview of checks, a chapter with details for red checks only, and the appendix.

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.



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.

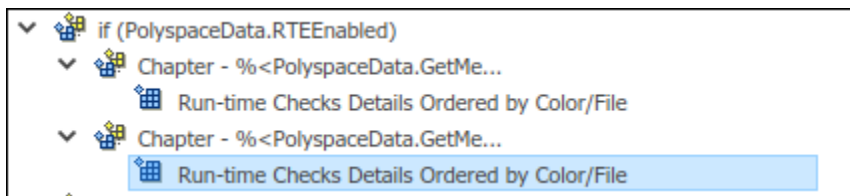
Add Chapter for Functional Design Errors

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`.



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 20-16

Software Quality with Polyspace Metrics

- “Code Quality Metrics” on page 21-2
- “Generate Code Quality Metrics” on page 21-9
- “View Code Quality Metrics” on page 21-12
- “Compare Metrics Against Software Quality Objectives” on page 21-15
- “View Trends in Code Quality Metrics” on page 21-20
- “Web Browser Requirements for Polyspace Metrics” on page 21-23
- “Elements in Custom Software Quality Objectives File” on page 21-24

Code Quality Metrics

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

- Provide your management a high-level overview of your code quality.
- Compare your code quality against predefined standards.
- Establish a process where you review in detail only those results that fail to meet standards.
- Track improvements or regression in code quality over time.

For each project or run, you can view the code quality metrics spread over four tabs, at project, file, and function level.

- The **Summary** tab provides a high-level overview of the verification results.
- The **Code Metrics** tab provides the details of the code complexity metrics in your results.

See “Code Metrics”.

- The **Coding rules** tab provides the details of the coding rule violations in your results.

See “Coding Standards”.

- The **Run-Time Checks** tab provides details of run-time checks in your results.

See “Run-Time Checks”.

If you turn on Software Quality Objectives, each tab also specifies how your project or run compares against those objectives. See “Compare Metrics Against Software Quality Objectives” on page 21-15.

Summary Tab

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

To see the results embedded in your source code, download the results from Polyspace Metrics to the user interface. For more information, see “Review Metrics for Particular Project or Run” on page 21-13.

Column Name		Description
Verification Status		Verification level completed. See <i>Verification level (-to)</i> .
Code Metrics	Files	Number of files in project.
	Lines of code	Number of lines of code, broken down by file.
Coding Rules	Confirmed Defects	Number of coding rule violations to which you assign a Severity of High, Medium or Low in the Polyspace user interface. See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.
	Violations	Total number of coding rule violations.

Column Name		Description
Run-Time Errors	Confirmed Defects	<p>Number of run-time checks to which you assign a Severity of High, Medium or Low in the Polyspace user interface.</p> <p>See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.</p>
	Run-Time Reliability	<p>A measure of your code quality, expressed as a percentage.</p> <p>The percentage is calculated as number of green and other justified checks divided by the total number of checks.</p> <p>To justify a check, in the Polyspace user interface, you must assign an appropriate Status. See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.</p>
Software Quality Objectives	Overall Status	<p>A status of PASS or FAIL based on whether your code satisfies the software quality objectives you specified.</p> <p>For more information, see “Compare Metrics Against Software Quality Objectives” on page 21-15.</p>
	Level	<p>The software quality objectives that you specify. You can either use a predefined set of objectives or specify your own objectives.</p> <p>See:</p> <ul style="list-style-type: none"> • “Software Quality Objectives” on page 16-60 • “Customize Software Quality Objectives” on page 21-17
	Review Progress	<p>A measure of your review progress, expressed as a percentage.</p> <p>The percentage is calculated as number of reviewed non-green checks and coding rule violations divided by the total number of non-green checks and rule violations.</p> <p>To review a check, in the Polyspace user interface, you must assign a Status. See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.</p>

Column Name		Description
	Justified Code Metrics	Percentage of code metrics threshold violations that you have justified. To justify a threshold violation, in the Polyspace user interface, you must assign an appropriate Status . See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.
	Justified Coding Rules	Percentage of coding rule violations that you have justified. To justify a rule violation, in the Polyspace user interface, you must assign an appropriate Status . See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.
	Justified Run-Time Errors	Percentage of run-time checks that you have justified. To justify a check, in the Polyspace user interface, you must assign an appropriate Status . See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

Code Metrics Tab

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

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

For more information, see “Code Metrics”.

Coding Rules Tab

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

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

Column Name		Description
Coding Rules	Confirmed Defects	Number of coding rule violations to which you assign a Severity of High, Medium, or Low in the Polyspace user interface. See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

Column Name		Description
	Justified	Number of coding rule violations that you have justified. To justify a rule violation, in the Polyspace user interface, assign an appropriate Status . See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.
	Violations	Total number of coding rule violations.
Software Quality Objectives	Quality Status	A status of PASS or FAIL based on whether your code satisfies the software quality objectives you specified. See “Compare Metrics Against Software Quality Objectives” on page 21-15.
	Level	The software quality objectives that you specify. You can either use a predefined set of objectives, or specify your own objectives. See: <ul style="list-style-type: none"> • “Software Quality Objectives” on page 16-60 • “Customize Software Quality Objectives” on page 21-17
	Review Progress	A measure of your review progress, expressed as a percentage. The percentage is calculated as the number of reviewed coding rule violations divided by the total number of violations. To mark a check as reviewed, in the Polyspace user interface, assign a Status to the check. See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

Run-Time Checks Tab

The **Run-Time Checks** tab lists the run-time checks in your project or run. For more information on the checks, see “Run-Time Checks”.

You can group the information in the columns by **Files** or **Run-Time Categories**.

Column Name	Description
Confirmed Defects	Number of run-time checks to which you assign a Severity of High, Medium, or Low in the Polyspace user interface. See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

Column Name		Description
Run-Time Selectivity		Percentage, calculated as the number of non-orange checks divided by the total number of checks.
Green Code	Checks	Number of green checks. See “Code Prover Result and Source Code Colors” on page 16-8.
Systematic Run-Time Errors (Red Checks)	Justified	Percentage of red checks that you have justified. To justify a check, in the Polyspace user interface, assign an appropriate Status . See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.
	Checks	Number of red checks. See “Code Prover Result and Source Code Colors” on page 16-8.
Unreachable Branches (Gray Checks)	Justified	Percentage of gray checks that you have justified. To justify a check, in the Polyspace user interface, assign an appropriate Status . See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.
	Checks	Number of gray checks. See “Code Prover Result and Source Code Colors” on page 16-8.
Other Run-Time Errors (Orange Checks)	Justified	Percentage of orange checks that you have justified. To justify a check, in the Polyspace user interface, assign an appropriate Status . See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.
	Checks	Number of orange checks. See “Code Prover Result and Source Code Colors” on page 16-8.
	Path-Related Issues	Number of orange checks that indicate a run-time error only on certain execution paths. See “Critical Orange Checks” on page 16-55.
	Bounded-Input Issues	Number of orange checks that indicate a run-time error only for certain inputs. You have specified external constraints on the inputs. See “Critical Orange Checks” on page 16-55.

Column Name		Description
	Unbounded-Input Issues	Number of orange checks that indicate a run-time error only for certain inputs. You have not specified any external constraints on the inputs. See “Critical Orange Checks” on page 16-55.
Non-terminating constructs	Justified	Percentage of non-terminating constructs that you have justified. To justify a check, in the Polyspace user interface, assign an appropriate Status . See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.
	Checks	Number of non-terminating constructs such as Non-terminating call or Non-terminating loop .
Software Quality Objectives	Quality Status	A status of PASS or FAIL based on whether your code satisfies the software quality objectives you specified. See “Compare Metrics Against Software Quality Objectives” on page 21-15.
	Level	The software quality objectives that you specify. You can either use a predefined set of objectives or specify your own objectives. See: <ul style="list-style-type: none"> • “Software Quality Objectives” on page 16-60 • “Customize Software Quality Objectives” on page 21-17
	Review Progress	A measure of your review progress, expressed as a percentage. The percentage is calculated as the number of reviewed checks divided by the total number of checks. To mark a check as reviewed, in the Polyspace user interface, assign a Status to the check. See “Address Polyspace Results Through Bug Fixes or Justifications” on page 18-2.

See Also

Related Examples

- “Generate Code Quality Metrics” on page 21-9
- “View Code Quality Metrics” on page 21-12
- “Compare Metrics Against Software Quality Objectives” on page 21-15

- “View Trends in Code Quality Metrics” on page 21-20

Generate Code Quality Metrics

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

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

For more information, see the Polyspace Code Prover Access documentation.

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

For more information, see “Code Quality Metrics” on page 21-2.

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

Upload Results to Polyspace Metrics After Remote Verification

If you perform verification on a remote cluster, you can specify that the results must be uploaded automatically to the Polyspace Metrics interface after verification.

To specify post-verification uploads using the Polyspace user interface, in your project configuration, select **Run Settings**. Along with **Run Code Prover analysis on a remote cluster**, select **Upload results to Polyspace Metrics**.

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

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

Command Line

To specify automatic uploads at the command line, use the option `Upload results to Polyspace metrics (-add-to-results-repository)`.

Upload Local Verification Results to Polyspace Metrics

If you perform a local verification on your desktop, you can upload your results to the Polyspace Metrics web interface. Even for remote verification, if you do not select **Upload results to**

Polyspace Metrics, after verification, the results are downloaded to your computer. You can upload them later.

To upload results from the Polyspace user interface, select a result in the **Project Browser** pane or open a result. Select **Metrics > Upload to Metrics**. Change the default project name and version number if needed.

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

Passwords

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

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

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

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

Command Line

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

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

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

See Also

`polyspace-results-repository`

Related Examples

- “View Code Quality Metrics” on page 21-12

- “Compare Metrics Against Software Quality Objectives” on page 21-15
- “View Trends in Code Quality Metrics” on page 21-20

View Code Quality Metrics

Before you can view software quality metrics, upload your results to the Polyspace Metrics repository. You can upload the results of a local verification or remote verification. For more information, see “Generate Code Quality Metrics” on page 21-9.

Open Metrics Interface

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

- If you have a local installation of Polyspace, select **Metrics > Open Metrics**.
- If you do not have a local installation, enter the following URL in a web browser:

protocol:// ServerName: PortNumber

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

To use HTTPS, set up the configuration file and the **Metrics configuration** preferences. For more information, see “Configure Web Server for HTTPS”.

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


View All Projects and Runs

On the Polyspace Metrics interface, you can view either all projects or all runs.

- On the **Projects** tab, view all projects.

On this tab, you can do the following:

Goal	Action
See number of project runs.	Hover your cursor over the project name.
Group projects together.	Right-click a project. Select Create Project Category . Drag projects to your new category.
Filter projects from display.	In the field below the Project column header, enter the name of the project you want.
Delete project from the Metrics repository.	Right-click the project. Select Delete Project from Repository .
Assign password to project.	Right-click the project. Select Change/Set Password .
See code quality metrics for all runs of project.	Click the project name. For more information, see “Review Metrics for Particular Project or Run” on page 21-13.

Tip If a new verification has been carried out for a project since your last visit, then on the **Projects** tab, the icon  appears before the project name.

- If a project has multiple runs, on the **Runs** tab, view the individual runs. To identify different runs of the same project, use the **Project** and **Version** column.

On this tab, you can do the following:

Goal	Action
Delete run from repository.	Right-click the run. Select Delete Run from Repository .
Assign password to run.	Right-click the run. Select Change/Set Password .
See runs between two specific dates.	Select the starting date in the From field and the end date in the To field.
See only the last n runs.	In the field Maximum number of runs , enter n .
See code quality metrics for a run.	Right-click the run. Select Go to Metrics Page . For more information, see “Review Metrics for Particular Project or Run” on page 21-13.
Download results of run to Polyspace user interface.	Click the run name.

Review Metrics for Particular Project or Run



If you select a project on the **Projects** tab or **Go to Metrics Page** for a run on the **Runs** tab, you can view the code quality metrics for the project or run. A summary of the metrics appears on the **Summary** tab.

If you want to compare the code quality metrics against standards you have previously defined, before reviewing your results, you can turn on quality objectives. For more information, see “Compare Metrics Against Software Quality Objectives” on page 21-15.

Otherwise, review the absolute values of code quality metrics on the **Summary** tab.

- 1 Select an entry on the **Summary** tab to open another tab with further details.
 - If you select an entry under the group **Code Metrics**, you can see your code complexity metrics on the **Code Metrics** tab.
 - If you select an entry under the group **Coding Rules**, you can see your coding rule violations on the **Coding Rules** tab.
 - If you select an entry under the group **Run-Time Errors**, you can see your run-time checks on the **Run-Time Checks** tab.

For example, in the following metrics, there are three red checks. Select the entry in the **Red** column to view the checks on the **Run-time Checks** tab.

Verification	Verification Status	Code Metrics		Coding Rules		Run-Time Errors					Review Progress	
		Files	Lines of Code	Confirmed Defects	Violations	Confirmed Defects	Run-Time Selectivity	Green	Red	Orange		Gray
 1.0 (2)	completed (PASS2)	1	125				91.8%	85	3	8	2	0.0%
 1.0 (1)	completed (PASS2)	1	125				91.8%	85	3	8	2	0.0%

For details on the columns, see “Code Quality Metrics” on page 21-2.

- 2 On the **Code Metrics**, **Coding Rules** or **Run-Time Errors** tabs, select an entry to download the result to the Polyspace user interface and follow any prompt from your web browser. If the results do not open automatically in the Polyspace interface, check the “Web Browser Requirements for Polyspace Metrics” on page 21-23.

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

- 3 In the Polyspace user interface, review the particular result, investigate the root cause in your source code, and assign review comments and justifications.

You can review the downloaded results or generate reports. If you generate a report, all results in the results file appear in the report (not just the downloaded results). To generate a filtered report, change the scope **Web checks** to another named filtered set, for instance, **All results**. Then, apply filters and generate the report. For more information, see “Generate Reports” on page 20-2.

- 4 To upload your comments and justifications to the Polyspace Metrics repository, select **Metrics > Upload to Metrics**.

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

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

-
- 5 After your review is over, in the Polyspace Metrics interface, click  to view updated metrics.

See Also

Related Examples

- “View Trends in Code Quality Metrics” on page 21-20

Compare Metrics Against Software Quality Objectives

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

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

For more information, see the Polyspace Code Prover Access documentation.

After generating and viewing metrics from your verification results, you can review the results in greater detail. You can download each result into the Polyspace user interface, investigate it in your source code and add review comments to them. For more information, see “View Code Quality Metrics” on page 21-12.

To focus your review, you can:

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

Apply Predefined Objectives to Metrics

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

- 1 Open the Polyspace Metrics interface. View the metrics for a project or a run on the **Summary** tab.


For more information, see “View Code Quality Metrics” on page 21-12.

- 2 From the **Quality Objectives** list in the upper left, select **ON**.


- A new group of **Software Quality Objectives** columns appears.
- In the **Overall Status** column, is the last used quality objective level to generate a status of **PASS** or **FAIL** for your results.
- In the **Level** column, you can see the quality objective level.

To change your quality objective level, in this column, select a cell. From the drop-down list, select a quality level. For more information, see “Software Quality Objectives” on page 16-60.


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

If the  icon appears next to the status, it means that Polyspace does not have sufficient information to compute the status. For instance, if you specify the level **SQ0-1**, but do not check for coding rule violations in your project, Polyspace cannot determine whether your project satisfies all the objectives specified in **SQ0-1**.

- 4 View further details for the entries which are marked red on the **Summary** tab. For example, if an entry on the **Code Metrics over Threshold** column is marked red, select it. You can see values of the code complexity metrics on the **Code Metrics** tab.
- 5 Review each code complexity metric, coding rule violation, or run-time error that caused your project to fail quality objectives. Fix your code or justify the errors or violations.

Tab	Action
Code Metrics	Note the entries that are red. Select each entry to download the code metric threshold violation to the Polyspace user interface. Review the violations and fix or justify it. If you justify a violation, you can upload your justifications to the Polyspace Metrics web dashboard. After justification, a red entry appears green with an  icon next to it.
Coding Rules	In the Justified column, note the entries that are red. Select each entry to download the coding rule violation to the Polyspace user interface. Review the violation and fix or justify it. If you justify a violation, you can upload your justifications to the Polyspace Metrics web dashboard. After justification, a red entry appears green in the Justified column.
Run-Time Checks	In the Justified columns, note the entries that are red. Select each entry to download the checks to the Polyspace user interface. Review the checks and fix or justify them. If you justify a check, you can upload your justifications to the Polyspace Metrics web dashboard. After justification, a red entry appears green in the Justified column.

For more information on the review process, see “Review Metrics for Particular Project or Run” on page 21-13.

- 6 After your review, in the Polyspace Metrics interface, click  to view the updated metrics. See if your project has an **Overall Status** of **PASS** because of your justifications.

If you change your code, to update the metrics, rerun your verification and upload the results to the Polyspace Metrics repository. If you have justifications in your previous results, import them to the new results before uploading the new results to the repository. To begin, select **Tools > Import Comments**.

Tip You can apply a quality objective to all files in a project or run. If you want to turn off quality objectives or apply different objectives for some files in your project, you can place them in a separate module.

To create a new module, press **Ctrl** and select the rows containing the files that you want to group. Right-click the selection, and select **Add to Module**. In the **Level** column for this module, select your quality objective from the drop-down list. The software applies this objective to all files in the module and determines an **Overall Status** of **PASS** or **FAIL** to the module.

Customize Software Quality Objectives

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

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

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

  <SQO ID="Custom-SQO-Level" ApplicableProduct="Code Prover"
      ApplicableProject="My_Project">
    <comf>20</comf>
    <path>80</path>
    <goto>0</goto>
    <vg>10</vg>
    <calling>5</calling>
    <calls>7</calls>
    <param>5</param>
    <stmt>50</stmt>
    <level>4</level>
    <return>1</return>
    <vocf>4</vocf>
    <ap_cg_cycle>0</ap_cg_cycle>
    <ap_cg_direct_cycle>0</ap_cg_direct_cycle>
    <Num_Unjustified_Violations>Custom_MISRA_Rules_Set
  </Num_Unjustified_Violations>
    <Num_Unjustified_Red>0</Num_Unjustified_Red>
    <Num_Unjustified_NT_Constructs>0
  </Num_Unjustified_NT_Constructs>
    <Num_Unjustified_Gray>0</Num_Unjustified_Gray>
    <Percentage_Proven_Or_Justified>
Custom_Runtime_Checks_Set</Percentage_Proven_Or_Justified>
  </SQO>

  <CodingRulesSet ID="Custom_MISRA_Rules_Set">
    <Rule Name="MISRA_C_5_2">0</Rule>
    <Rule Name="MISRA_C_17_6">0</Rule>
  </CodingRulesSet>

  <RuntimeChecksSet ID="Custom_Runtime_Checks_Set">
    <Check Name="OBAI">80</Check>
    <Check Name="IDP">60</Check>
  </RuntimeChecksSet>

</MetricsDefinitions>
```

You can use this file for both Bug Finder and Code Prover results. For information on the XML elements specific to Bug Finder, see “Compare Metrics Against Software Quality Objectives” (Polyspace Bug Finder).

- 2 Save this XML file in the folder where remote analysis data is stored, for example, C:\Users\JohnDoe\AppData\Roaming\Polyspace_RLData.s.

If you want to change the folder location, select **Metrics > Metrics and Remote Server Settings**.

- 3 Modify the content of this file to specify the project name and your own quality thresholds. For more information, see “Elements in Custom Software Quality Objectives File” on page 21-24.
 - a To make the quality level `Custom-SQO-Level` applicable to a certain project, replace the value of the `ApplicableProject` attribute with the project name.

If you want the quality objectives to apply to all projects, use `ApplicableProject=""`.

- b For specifying coding rules, begin the rule name with the appropriate string followed by the rule number. Use `_` instead of a decimal point in the rule number. To specify directives, begin specifying the directive number with `D_`, for instance, `MISRA_C3_D4_6`, for MISRA C: 2012 directive 4.6.

Rule	String	Rule numbers
MISRA C: 2004	MISRA_C_	“MISRA C:2004 and MISRA AC AGC Coding Rules” on page 13-3
MISRA C: 2012	MISRA_C3_	“MISRA C:2012 Directives and Rules”
MISRA C++	MISRA_Cpp_	“MISRA C++:2008 Rules”
JSF C++	JSF_Cpp_	“JSF C++ Coding Rules” on page 13-44
Custom coding rules	Custom_	“Custom Coding Rules” (Polyspace Bug Finder)

- c For specifying checks, use the appropriate check acronym. For more information, see “Short Names of Code Prover Run-Time Checks” on page 18-12.
 - d For specifying code metrics, use the code metric acronym. See “Short Names of Code Complexity Metrics” on page 18-14.
- 4 After you have made your modifications, in the Polyspace Metrics interface, open the metrics for your project. From the **Quality Objectives** list in the upper left, select ON.
- 5 On the **Summary** tab, select an entry in the **Level** column. For the project name that you specified, your new quality objective **Custom-SQO-Level** appears in the drop-down list.
- 6 Select your new quality objective.

The software compares the thresholds you had specified against your results and updates the **Overall Status** column with **PASS** or **FAIL**.

- 7 To define another set of custom quality objectives, add the following content to the `Custom-SQO-Definitions.xml` file:

```
<SQO ID="Custom-SQO-Level_2" ParentID="Custom-SQO-Level"
  ApplicableProduct="Code Prover"
  ApplicableProject="My_Project">
  ...
</SQO>
```

Here:

- ID represents the name of the new set.

You cannot have the same values of ID and `ApplicableProject` for two different sets of quality objectives. For example, if you use an ID value of `Custom-SQ0-Level` for two different sets, and an `ApplicableProject` value of `My_Project` for one set and `My_Project` or `" "` for the other, you see the following error:

```
The SQ0 level 'Custom-SQ0-Level' is multiply defined.
```

- `ParentID` specifies another level from which the current level inherits its quality objectives. In the preceding example, the level `Custom-SQ0-Level_2` inherits its quality objectives from the level `Custom-SQ0-Level`.

If you do not want to inherit quality objectives from another level, omit this attribute.

- `...` represents the additional quality thresholds that you specify for the level `Custom-SQ0-Level_2`.

The quality thresholds that you specify override the thresholds that `Custom-SQ0-Level_2` inherits from `Custom-SQ0-Level`. For instance, if you specify `<goto>1</goto>`, this overrides the threshold specification `<goto>0</goto>` of `Custom-SQ0-Level`.

See Also

Related Examples

- “View Trends in Code Quality Metrics” on page 21-20

View Trends in Code Quality Metrics

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

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

For more information, see the Polyspace Code Prover Access documentation.

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

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

- 1 Open the Polyspace Metrics interface.

For more information, see “Open Metrics Interface” on page 21-12.

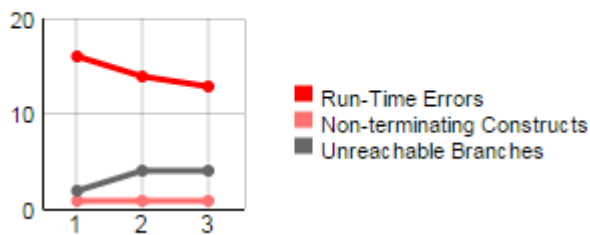
- 2 On the **Projects** tab, select the project for which you want to view trends.

The code quality metrics for all versions of the project appear on the **Summary**, **Code Metrics**, **Coding Rules**, and **Run-Time Checks** tabs. For example, the figure shows the **Summary** tab displaying three versions of a project.

Verification	Verification Status	Code Metrics		Coding Rules		Run-Time Errors					Review Progress	
		Files	Lines of Code	Confirmed Defects	Violations	Confirmed Defects	Run-Time Selectivity	Green	Red	Orange		Gray
1.0 (3)	completed (PASS2)	7	955				95.5%	717	14	35	4	0.0%
1.0 (2)	completed (PASS2)	7	955				95.3%	716	15	36	4	0.0%
1.0 (1)	completed (PASS2)	7	955				95.2%	694	17	36	2	0.0%

In addition, you can see a graphical view of the trends on each tab. For example, the figure shows the trend in **Run-Time Findings** over three versions of a project.

Run-Time Findings



- 3 To compare two versions of the same project:

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

On each tab, new columns appear and existing columns display improvement or regression in a metric. For example, in the figure below, you see a new **All Metrics Trend** column that appears on the **Summary** tab. This column describes how the metrics in the **Run-Time Errors** group compare over two versions of a project. The number of red checks decreased by 3 and the number of gray checks increased by 2. Because the decrease in red checks is an improvement and the increase in gray checks is a regression, you see:

- A ▲ in the **Red** column
- A ▼ in the **Gray** column
- A mixed ▲▼ in the **All Metrics Trend** column.

Run-Time Errors						
Confirmed Defects	Run-Time Selectivity	Green	Red	Orange	Gray	All Metrics Trend
	95.5% (+0.3%)	717 (+23)▲	14 (-3)▲	35 (-1)▲	4 (+2)▼	▲▼
	99.6% (+0.0%)	234 (+1)▲		1		▲
	64.7%	22		12		
	92.3%	70		6	2	
	97.9%	138	2	3		
	100.0% (+1.0%)	101 (+22)▲	5 (-3)▲	0 (-1)▲	2 (+2)▼	▲▼
	67.9%	18	1	9		
	95.4%	61	1	3		
	98.7%	73	5	1		

- 4 To see only the new findings in a version compared to a previous version:
 - a In the **From** and **To** lists on the upper left of the web dashboard, select the two versions that you want to compare.
 - b Select the **New Findings Only** box.

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

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

See Also

Related Examples

- “Code Quality Metrics” on page 21-2

Web Browser Requirements for Polyspace Metrics

To use Polyspace Metrics, install Java®, version 1.4 or later on your computer.

Polyspace Metrics supports these web browsers:

- Internet Explorer® version 7.0, or later
- Firefox® version 3.6, or later
- Google® Chrome version 12.0, or later
- Safari for Mac version 6.1.4 and 7.0.4

Additional Considerations

- With certain web browsers such as Google Chrome, you are prompted to download or open a file called `psapplet.jsp`. You must configure your system to open `.jsp` files using the Java Web Start Launcher (`javaws`) binary. You can find this binary in your Java installation folder.

For instance, if you download `psapplet.jsp` on a Windows system, to open and view your Polyspace results:

- 1 Right-click `psapplet.jsp` and select **Properties**.
 - 2 Next to **Opens With**: select the Java Web Start Launcher (`javaws`) binary. If the binary is not in the default list of options, browse to your Java installation folder, for example `C:\Program Files\Java\jre1.8.0_161\bin`.
 - 3 After you apply your changes, double-click `psapplet.jsp` to open it.
- If your computer uses the Linux operating system, manually install the required Java plug-in for the Firefox web browser:

- 1 Create a `plugins` folder under the `.mozilla` folder in your home directory:

```
mkdir ~/.mozilla/plugins
```

- 2 From this folder, create a symbolic link to the Java plug-in, which is available in the Java Runtime Environment folder.

For instance, to use the Java Runtime Environment of your MATLAB installation, enter these commands:

```
cd ~/.mozilla/plugins
ln -s MATLAB_Install/sys/java/jre/glnxa64/jre/lib/amd64/libnjp2.so
```

- If you download results using Internet Explorer 11, it may take a minute or two to open the Java plug-in and load the Polyspace interface.

Elements in Custom Software Quality Objectives File

The following tables list the XML elements that can be added to the custom SQO file. The content of each element specifies a threshold against which the software compares verification results. For each element, the table lists the metric to which the threshold applies. Here, HIS refers to the Hersteller Initiative Software.

For information on custom SQOs, see “Customize Software Quality Objectives” on page 21-17.

HIS Metrics

Element	Metric
comf	Comment Density
path	Number of Paths
goto	Number of Goto Statements
vg	Cyclomatic Complexity
calling	Number of Calling Functions
calls	Number of Called Functions
param	Number of Function Parameters
stmt	Number of Instructions
level	Number of Call Levels
return	Number of Return Statements
vocf	Language Scope
ap_cg_cycle	Number of Recursions
ap_cg_direct_cycle	Number of Direct Recursions
Num_Unjustified_Violations	Number of unjustified violations of MISRA C rules specified by entries under the element CodingRulesSet
Num_Unjustified_Red	Number of unjustified red checks
Num_Unjustified_NT_Constructs	Number of unjustified Non-terminating call and Non-terminating loop checks
Num_Unjustified_Gray	Number of unjustified gray Unreachable code checks
Percentage_Proven_Or_Justified	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.

Non-HIS Metrics

Element	Description of metric
fco	Estimated Function Coupling

Element	Description of metric
flin	Number of Lines Within Body
fxln	Number of Executable Lines
ncalls	Number of Call Occurrences
pshv	Number of Protected Shared Variables
unpshv	Number of Potentially Unprotected Shared Variables

Troubleshooting in Polyspace Code Prover

- “View Error Information When Analysis Stops” on page 22-3
- “Troubleshoot Compilation and Linking Errors” on page 22-6
- “Reduce Memory Usage and Time Taken by Polyspace Analysis” on page 22-10
- “Understand Verification Results” on page 22-15
- “Contact Technical Support About Issues with Running Polyspace” on page 22-18
- “Polyspace Cannot Find the Server” on page 22-21
- “Job Manager Cannot Write to Database” on page 22-22
- “Compiler Not Supported for Project Creation from Build Systems” on page 22-23
- “Slow Build Process When Polyspace Traces the Build” on page 22-29
- “Check if Polyspace Supports Build Scripts” on page 22-30
- “Troubleshooting Project Creation from MinGW Build” on page 22-32
- “Troubleshooting Project Creation from Visual Studio Build” on page 22-33
- “Error Processing Macro with Semicolon in Build System” on page 22-34
- “Could Not Find Include File” on page 22-35
- “Conflicting Universal Unique Identifiers (UUIDs)” on page 22-37
- “Data Type Not Recognized” on page 22-38
- “Undefined Identifier Error” on page 22-40
- “Unknown Function Prototype Error” on page 22-43
- “Error Related to #error Directive” on page 22-44
- “Large Object Error” on page 22-45
- “Errors Related to Generic Compiler” on page 22-47
- “Errors Related to Keil or IAR Compiler” on page 22-48
- “Errors Related to Diab Compiler” on page 22-49
- “Errors Related to Green Hills Compiler” on page 22-51
- “Errors Related to TASKING Compiler” on page 22-53
- “Errors from In-Class Initialization (C++)” on page 22-55
- “Errors from Double Declarations of Standard Template Library Functions (C++)” on page 22-56
- “Errors Related to GNU Compiler” on page 22-57
- “Errors Related to Visual Compilers” on page 22-58
- “Conflicting Declarations in Different Translation Units” on page 22-60
- “Errors from Conflicts with Polyspace Header Files” on page 22-65
- “C++ Standard Template Library Stubbing Errors” on page 22-66
- “Lib C Stubbing Errors” on page 22-67

- “Errors from Using Namespace std Without Prefix” on page 22-69
- “Errors from Assertion or Memory Allocation Functions” on page 22-70
- “Eclipse Java Version Incompatible with Polyspace Plug-in” on page 22-71
- “Reasons for Unchecked Code” on page 22-72
- “Source Files or Functions Not Displayed in Results List” on page 22-78
- “Coding Standard Violations Not Displayed” on page 22-81
- “Incorrect Behavior of Standard Library Math Functions” on page 22-83
- “Insufficient Memory During Report Generation” on page 22-84
- “Errors with Temporary Files” on page 22-85
- “Error or Slow Runs from Disk Defragmentation and Anti-virus Software” on page 22-87
- “SQLite I/O Error” on page 22-89
- “License Error -4,0” on page 22-90

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.

For information on why Polyspace fails to compile your code despite successful compilation with your compiler, see “Troubleshoot Compilation and Linking Errors” on page 22-6.

View Error Information in User Interface

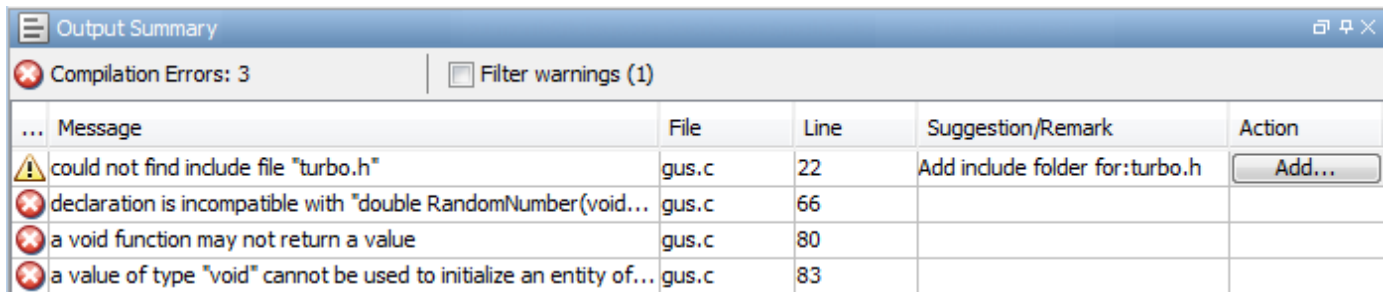
- 1 View the errors on the **Output Summary** tab.

The messages on this tab appear with the following icons.

Icon	Meaning
	Error that blocks analysis. For instance, the analysis cannot find a variable declaration or definition and therefore cannot determine the variable type.
	Warning about an issue that does not block analysis by itself, but could be related to a blocking error. For instance, the analysis cannot find an include file that is <code>#include-d</code> in your code. The issue does not block the analysis by itself, but if the include file contains the definition of a variable that you use in your source code, you can face an error later.
	Additional information about the analysis.

- 2 To diagnose and fix each error, you can do the following:
 - To see further details about the error, select the error message. The details appear in a **Detail** window below the **Output Summary** tab.
 - To open the source code at the line containing the error, double-click the message.
- 3 If you enable the Compilation Assistant, to fix an error, you can perform certain actions on the **Output Summary** tab.

The following figure shows an error due to a missing include file `turbo.h`. You can add the missing file by clicking the **Add** button on the **Output Summary** tab.



To turn on the Compilation Assistant, select **Tools > Preferences**. On the **Project and Results Folder** tab, select **Use Compilation Assistant**.

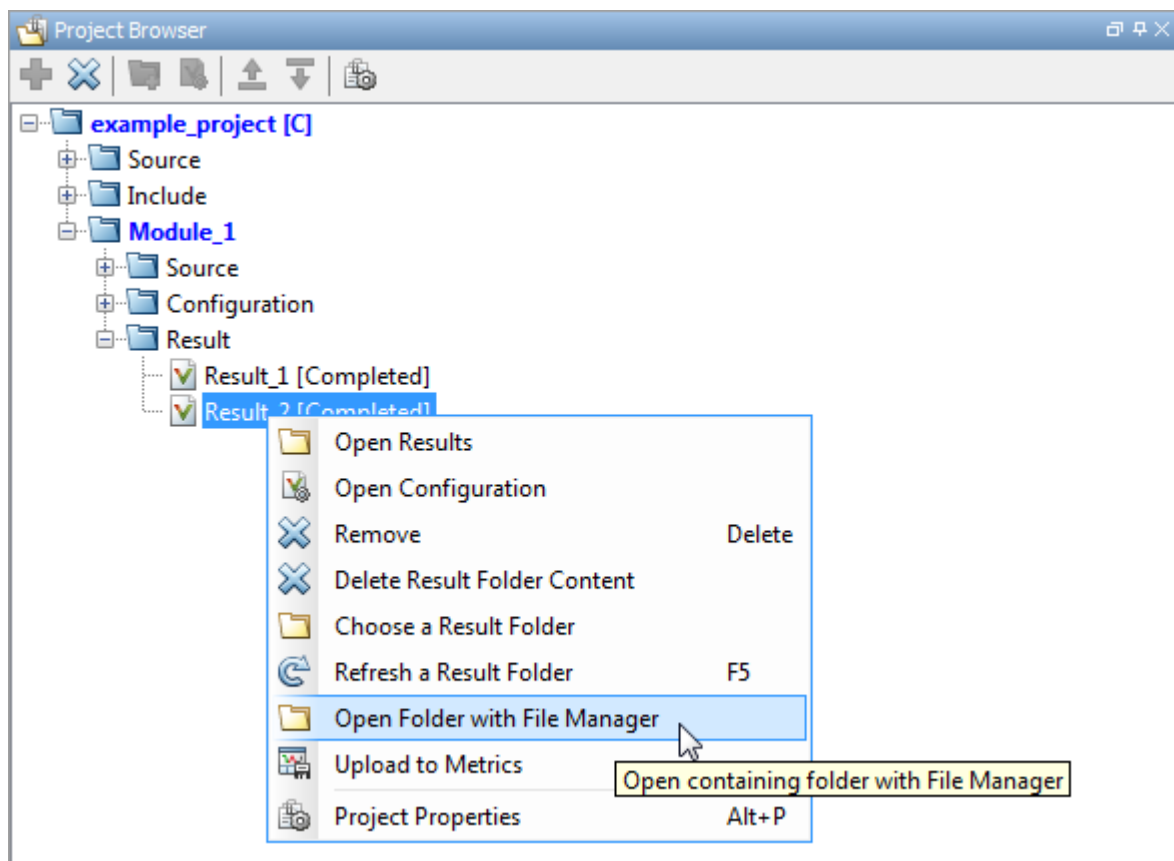
The Compilation Assistant is disabled if you specify the option `Verify files independently (-unit-by-unit)` or `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

Tip To search the error messages for a specific term, on the **Search** pane, enter your search term. From the drop down list on this pane, select **Output Summary** or **Run Log**. If the **Search** pane is not open by default, select **Windows > Show/Hide View > Search**.

View Error Information in Log File

You can view errors directly in the log file. The log file is in your results folder. To open the log file:

- 1 Right-click the result folder name on the **Project Browser** pane. From the context menu, select **Open Folder with File Manager**.



- 2 Open the log file, `Polyspace_R20##n_ProjectName_date-time.log`
- 3 To view the errors, scroll through the log file, starting at the end and working backward.

The following example shows sample log file information. The error has occurred because 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

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.

Type	Message	File	Line	Col
	Verification running			
			Elapsed time: 00:00:08	
			Total elapsed time: 00:00:08	
	C verification starts at Mon Dec 07 16:48:05 2015			
	6 core(s) detected but the verification uses 4 core(s).			

Compilation Phase

Type	Message	File	Line	Col
	C verification starts at Thu Dec 17 22:26:17 2015			
	6 core(s) detected but the verification uses 4 core(s).			
	identifier "x" is undefined	my_file.c	1	
	Failed compilation.			
	Verifier has detected compilation error(s) in the code.			
	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.

If you turn on the **Compilation Assistant** and run verification in the user interface, for most compilation errors, you receive suggestions in the **Output Summary** pane that you can apply in one click. See “View Error Information When Analysis Stops” on page 22-3.

Otherwise, you can manually adjust your analysis options. To get past compilation issues, use 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.
<ul style="list-style-type: none"> • 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.

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 22-18.

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 “Errors from Conflicts with Polyspace Header Files” on page 22-65.

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 “Errors from Assertion or Memory Allocation Functions” on page 22-70.

Reduce Memory Usage and Time Taken by Polyspace Analysis

In this section...

“Issue” on page 22-10

“Possible Cause: Temporary Folder on Network Drive” on page 22-10

“Possible Cause: Anti-Virus Software” on page 22-10

“Possible Cause: Large and Complex Application” on page 22-11

“Possible Cause: Too Many Entry Points for Multitasking Applications” on page 22-13

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” on page 1-12.

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 “Error or Slow Runs from Disk Defragmentation and Anti-virus Software” on page 22-87.

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.
 - 1 Run a quick initial verification with the lowest precision and verification level. See Precision level (-0) and Verification level (-to).
 - 2 Select **Tools > Run Modularize**.

The software displays a plot between the number of modules and the number of cross-module variables/functions or the complexity of each module. Using this plot for guidance, choose the number of modules into which the project must be partitioned. You have to make a compromise between the cross-module dependence and the complexity of each module.

- 3 Select the vertical gray region on the plot corresponding to the number of modules you have chosen. The software splits your project into that many modules.

You can modularize your application from the command line. Use the `polyspace-modularize` command in `polyspaceroot\polyspace\bin`. For more information on the command, use the `-h` option.

- 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 (-O)`.
- 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” on page 12-44: 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)
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 “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: Important use of shared variables have been detected,
|   verification carry on but to avoid scaling issues
|   it roughly approximates shared variables values.
|   You may consider adding -force-refined-shared-variables-analysis
|                                   option to improve results
```

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

Instead of using the option `-force-refined-shared-variables-analysis` to retain the precise analysis, 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 11-16.

See Also

External Websites

- Resolving Scaling Problems in Code Prover

Understand Verification Results

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 16-8.

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.

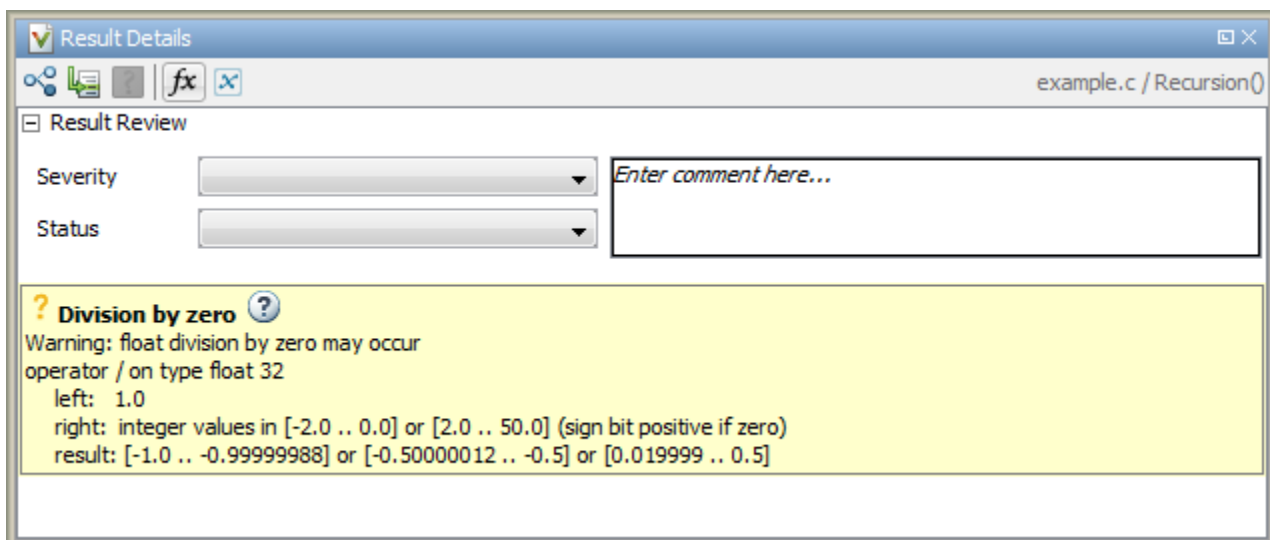
Solution

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 “Code Prover Run-Time Checks” on page 16-13.

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 22-18.

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, run the following command, replacing *polyspaceroot* with your Polyspace installation folder:
 - UNIX — `polyspaceroot/polyspace/bin/polyspace-code-prover -ver`
 - Windows — `polyspaceroot\polyspace\bin\polyspace-code-prover -ver`

Provide Information About the Issue

Depending on the issue, provide appropriate artifacts to help Technical Support understand and reproduce the issue.

Compilation Errors

If you face compilation issues with your project, see “Troubleshoot Compilation Errors”. If you are still having issues, contact technical support with the following information:

- The analysis log.

The analysis log is a text file generated in your results folder and titled `Polyspace_version_project_date_time.log`. It contains the error message, the options used for the analysis and other relevant information.

- The source files related to the compilation error or the complete results folder if possible.

If you cannot provide the source files:

- Try to provide a screenshot of the source code section that causes the compilation issue.
- Try to reproduce the issue with a different code. Provide that code to technical support.

Errors in Project Creation from Build Systems

If you face errors in creating a project from your build system, see “Troubleshoot Project Creation”.

If you are still having issues, contact technical support with debug information. To provide the debug information:

- 1 Run `polyspace-configure` at the command line with the option `-easy-debug`. For instance:
`polyspace-configure options -easy-debug pathToFolder buildCommand`

Here:

- `options` is the list of `polyspace-configure` options that you typically use.
- `buildCommand` is the build command that you use, for instance, `make`.
- `pathToFolder` is the folder where you want to store debug information, for instance, `C:\Temp\BuildLogs`. After a `polyspace-configure` run, the path provided contains a zipped file ending with `pscfg-output.zip`. The zipped file contains debug information only and does not contain source files traced in the build.

Make sure that you do not use the option `-verbose` or `-silent` after `-easy-debug`. These options reduce or modify the information logged and might make debugging difficult.

- 2 Send this zipped file ending with `pscfg-output.zip` to MathWorks Technical Support for further debugging.

You can also create the zipped file with debug information during every `polyspace-configure` run by creating an environment variable `PS_CONFIGURE_OPTIONS` and setting its value to:

`-easy-debug pathToFolder`

where `pathToFolder` is the folder where you want to store debug information.

Verification Result

If you are having trouble understanding a result, see 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 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.

Polyspace Cannot Find the Server

Message

```
Error: Cannot instantiate Polyspace cluster
| Check the -scheduler option validity or your default cluster profile
| Could not contact an MJS lookup service using the host computer_name.
  The hostname, computer_name, could not be resolved.
```

Possible Cause

Polyspace uses information provided in the preferences of a Polyspace desktop product to locate the server. If this information is incorrect, the software cannot locate the server.

Solution

Open the user interface of the Polyspace desktop product. Check if the server information provided is correct.

- 1 Select **Tools > Preferences**.
- 2 Select the **Server Configuration** tab. Check your server information.

For instance, the entry in **Job scheduler host name** must match the host name of the computer that forms the head node of the MATLAB Parallel Server cluster. For more information, see “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

Job Manager Cannot Write to Database

Message

Unable to write data to the job manager database

Possible Cause

If the computer that forms the head node of the MATLAB Parallel Server cluster cannot send data to the client computer, you see this error. The most likely reasons for the remote computer being unable to connect to the client computer are:

- Firewalls do not allow traffic from the MATLAB Job Scheduler to the client.
- The MATLAB Job Scheduler cannot resolve the short hostname of the client computer.

Workaround

Add localhost IP to configuration.

- 1 In the user interface of the Polyspace desktop products, select **Tools > Preferences**.
- 2 On the **Server Configuration** tab, in the **Localhost IP address** field, enter the IP address of your local computer.

To retrieve your IP address:

- Windows
 - 1 Open **Control Panel > Network and Sharing Center**.
 - 2 Select your active network.
 - 3 In the Status window, click **Details**. Your IP address is listed under **IPv4 address**.
- Linux — Run the `ifconfig` command and find the `inet addr` corresponding to your network connection.
- Mac — Open **System Preferences > Network**.

See Also

Related Examples

- “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”
- “Connection Problems Between the Client and MATLAB Job Scheduler” (Parallel Computing Toolbox)

Compiler Not Supported for Project Creation from Build Systems

Issue

Your compiler is not supported for automatic project creation from build commands.

Cause

For automatic project creation from your build system, your compiler configuration must be available to Polyspace. Polyspace provides a compiler configuration file only for certain compilers.

For information on which compilers are supported, see “Requirements for Project Creation from Build Systems” on page 9-15.

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 22-27.
- 2 Save the file as *my_compiler.xml*. *my_compiler* can be a name that helps you identify the file.

To edit the file, save it outside the installation folder. After you have finished editing, you must copy the file back to *polyspaceroot\polyspace\configure\compiler_configuration*.

- 3 Edit the contents of the file to represent your compiler. Replace the entries between the XML elements with appropriate content.
- 4 After saving the edited XML file to *polyspaceroot\polyspace\configure\compiler_configuration*, create a project automatically using your build command.

If you see errors, for instance, compilation errors, contact MathWorks Technical Support. After tracing your build command, the software compiles certain files using the compiler specifications detected from your configuration file and build command. Compilation errors might indicate issues in the configuration file.

Tip To quickly see if your compiler configuration file works, run the automatic project setup on a sample build that does not take much time to complete. After you have set up a project with your compiler configuration file, you can use this file for larger builds.

Elements of Compiler Configuration File

The following table lists the XML elements in the compiler configuration file file with a description of what the content within the element represents.

XML Element	Content Description	Content Example for GNU C Compiler
<pre><compiler_names><name> ... </name></compiler_names></pre>	<p>Name of the compiler executable. This executable transforms your .c files into object files. You can add several binary names, each in a separate <name>...</name> element. The software checks for each of the provided names and uses the compiler name for which it finds a match.</p> <p>You must not specify the linker binary inside the <name>...</name> elements.</p> <p>If the name that you specify is present in an existing compiler configuration file, an error occurs. To avoid the error, use the additional option <code>-compiler-config my_compiler.xml</code> when tracing the build so that the software explicitly uses your compiler configuration file.</p>	<ul style="list-style-type: none"> • gcc • gpp
<pre><include_options><opt> ... </opt></include_options></pre>	<p>The option that you use with your compiler to specify include folders.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-I</p>
<pre><system_include_options> <opt> ... </opt> </system_include_options></pre>	<p>The option that you use with your compiler to specify system headers.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-isystem</p>
<pre><preinclude_options><opt> ... </opt></preinclude_options></pre>	<p>The option that you use with your compiler to force inclusion of a file in the compiled object.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-include</p>

XML Element	Content Description	Content Example for GNU C Compiler
<pre><define_options><opt> ... </opt></define_options></pre>	<p>The option that you use with your compiler to predefine a macro.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-D</p>
<pre><undefine_options><opt> ... </opt></undefine_options></pre>	<p>The option that you use with your compiler to undo any previous definition of a macro.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-U</p>
<pre><semantic_options><opt> ... </opt></semantic_options></pre>	<p>The options that you use to modify the compiler behavior. These options specify the language settings to which the code must conform.</p> <p>You can use the <code>isPrefix</code> attribute to specify multiple options that have the same prefix and the <code>numArgs</code> attribute to specify options with multiple arguments. For instance:</p> <ul style="list-style-type: none"> • Instead of <pre><opt>-m32</opt> <opt>-m64</opt></pre> <p>You can write <code><opt isPrefix="true">-m</opt></code>.</p> • Instead of <pre><opt>-std=c90</opt> <opt>-std=c99</opt></pre> <p>You can write <code><opt numArgs="1">-std</opt></code>. If your makefile uses <code>-std=c90</code> instead of <code>-std=c90</code>, this notation also supports that usage.</p> 	<ul style="list-style-type: none"> • -ansi • -std =C90 • -std =c++11 • -fun signed -char

XML Element	Content Description	Content Example for GNU C Compiler
<code><compiler> ... </compiler></code>	<p>The Polyspace compiler option that corresponds to or closely matches your compiler. The content of this element directly translates to the option Compiler in your Polyspace project or options file.</p> <p>For the complete list of compilers available, see <code>Compiler (-compiler)</code>.</p>	gnu4.7
<code><preprocess_options_list></code> <code><opt> ... </opt></code> <code></preprocess_options_list></code>	<p>The options that specify how your compiler generates a preprocessed file.</p> <p>You can use the macro <code>\$(OUTPUT_FILE)</code> if your compiler does not allow sending the preprocessed file to the standard output. Instead it defines the preprocessed file internally.</p>	<p>-E</p> <p>For an example of the <code>\$(OUTPUT_FILE)</code> macro, see the existing compiler configuration file <code>cl2000.xml</code>.</p>
<code><preprocessed_output_file> ... </preprocessed_output_file></code>	<p>The name of file where the preprocessed output is stored.</p> <p>You can use the following macros when the name of the preprocessed output file is adapted from the source file:</p> <ul style="list-style-type: none"> • <code>\$(SOURCE_FILE)</code>: Source file name • <code>\$(SOURCE_FILE_EXT)</code>: Source file extension • <code>\$(SOURCE_FILE_NO_EXT)</code>: Source file name without extension <p>For instance, use <code>\$(SOURCE_FILE_NO_EXT).pre</code> when the preprocessor file name has the same name as the source file, but with extension <code>.pre</code>.</p>	<p>For an example of this element, see the existing compiler configuration file <code>xc8.xml</code>.</p>
<code><src_extensions><ext> ... </ext></src_extensions></code>	<p>The file extensions for source files.</p>	<ul style="list-style-type: none"> • c • cpp • c++

XML Element	Content Description	Content Example for GNU C Compiler
<pre><obj_extensions><ext> ... </ext></obj_extensions></pre>	The file extensions for object files.	
<pre><precompiled_header_extensions> ... </precompiled_header_extensions></pre>	The file extensions for precompiled headers (if available).	
<pre><polyspace_extra_options_list> <opt> ... </opt> <opt> ... </opt> </polyspace_extra_options_list></pre>	<p>Additional options that are used for the subsequent analysis.</p> <p>For instance, to avoid compilation errors in the subsequent analysis due to non-ANSI extension keywords, enter <code>-D keyword=value</code>, for example:</p> <pre><polyspace_extra_options_list> <opt>-D MACR01</opt> <opt>-D MACR02=VALUE</opt> </polyspace_extra_options_list></pre> <p>For more information, see Preprocessor definitions (-D).</p>	

Mapping Between Existing Configuration Files and Compiler Names

Select the configuration file in `polyspaceroot\polyspace\configure\compiler_configuration\` that most closely resembles the configuration of your compiler. Use the following table to map compilers to their configuration files.

Compiler Name	Vendor	XML File
ARM®	ARM Keil	armcc.xml
		armclang.xml
Visual C++	Microsoft	cl.xml
Clang	Not applicable	clang.xml
CodeWarrior	NXP	cw_ppc.xml
		cw_s12z.xml
cx6808	Cosmic	cx6808.xml
Diab	Wind River	diab.xml
gcc	Not applicable	gcc.xml
Green Hills	Green Hills Software	ghs_arm.xml
		ghs_arm64.xml
		ghs_i386.xml
		ghs_ppc.xml

Compiler Name	Vendor	XML File
		ghs_rh850.xml
		ghs_tricore.xml
IAR Embedded Workbench	IAR	iar.xml
		iar-arm.xml
		iar-avr.xml
		iar-msp430.xml
		iar-rh850.xml
		iar-rl78.xml
Renesas	Renesas	renesas-rh850.xml
		renesas-rl78.xml
		renesas-rx.xml
TASKING®	Altium	tasking.xml
		tasking-166.xml
		tasking-850.xml
		tasking-arm.xml
Tiny C	Not applicable	tcc.xml
TM320 and its derivatives	Texas Instruments	ti_arm.xml
		ti_c28x.xml
		ti_c6000.xml
		ti_msp430.xml
xc8 (PIC)	Microchip	xc8.xml

Slow Build Process When Polyspace Traces the Build

Issue

In some cases, your build process can run slower when Polyspace traces the build.

Cause

Polyspace caches information in files stored in the system temporary folder, such as `C:\Users\User_Name\AppData\Local\Temp`, in Windows. Your build can take a long time to perform read/write operations to this folder. Therefore, the overall build process is slow.

Solution

You can work around the slow build process by changing the location where Polyspace stores cache information. For instance, you can use a cache path local to the drive from which you run build tracing. To create and use a local folder `ps_cache` for storing cache information, use the advanced option `-cache-path ./ps_cache`.

- If you trace your build from the Polyspace user interface, enter this flag in the field **Add advanced configure options**.
- If you trace your build from the DOS/ UNIX or MATLAB command line, use this flag with the `polyspace-configure` command.

For more information, see `polyspace-configure`.

Check if Polyspace Supports Build Scripts

Issue

This topic is relevant only if you are creating a Polyspace project in Windows from your build scripts.

When Polyspace traces your build script in a Windows console application other than `cmd.exe`, the command fails. However, the build command by itself executes to completion.

For instance, your build script executes to completion from the Cygwin shell. However, when Polyspace traces the build, the build script throws an error.

Possible Cause

When you launch a Windows console application, your environment variables are appropriately set. Alternate console applications such as the Cygwin shell can set your environment differently from `cmd.exe`.

Polyspace attempts to trace your build script with the assumption that the script runs to completion in `cmd.exe`. Therefore, even if your script runs to completion in the alternate console application, when Polyspace traces the build, the script can fail.

Solution

Make sure that your build script executes to completion in the `cmd.exe` interface. If the build executes successfully, create a wrapper `.bat` file around your script and trace this file.

For instance, before you trace a build command that executes to completion in the Cygwin shell, do one of the following:

- Launch the Cygwin shell from `cmd.exe` and then run your build script. For instance, if you use a script `build.sh` to build your code, enter the following command at the DOS command line:

```
cmd.exe /C "C:\cygwin64\bin\bash.exe" -c build.sh
```

- Find the full path to your build script and then run this script from `cmd.exe`.

For instance, enter the following command at the DOS command line:

```
cmd.exe /C path_to_script
```

`path_to_script` is the full path to your build script. For instance, `C:\my_scripts\build.sh`.

If the steps do not execute to completion, Polyspace cannot trace your build.

If the steps complete successfully, trace the build command after launching it from `cmd.exe`. For instance, on the command-line, do the following to create a Polyspace options file.

- 1 Enter your build commands in a `.bat` file.

```
rem @echo off
cmd.exe /C "C:\cygwin64\bin\bash.exe" -c build.sh
```

Name the file, for instance, `launching.bat`.

2 Trace the build commands in the .bat file and create a Polyspace options file.

```
"C:\Program Files\MATLAB\R2017b\polyspace\bin\polyspace-configure.exe"  
-output-options-file myOptions.txt launching.bat
```

You can now run `polyspace-code-prover` on the options file.

Troubleshooting Project Creation from MinGW Build

Issue

You create a project from a MinGW build, but get an error when running an analysis on the project. The error message comes from using one of these keywords: `__declspec`, `__cdecl`, `__fastcall`, `__thiscall` or `__stdcall`.

Cause

When you create a project from a MinGW build, the project uses a GNU compiler. Polyspace does not recognize these keywords for the GNU compilers.

Solution

Replace these keywords with equivalent keywords just for the purposes of analysis.

Before analysis, for the option Preprocessor definitions (-D), enter:

- `__declspec(x)=__attribute__((x))`
- `__cdecl=__attribute__((__cdecl__))`
- `__fastcall=__attribute__((__fastcall__))`
- `__thiscall=__attribute__((__thiscall__))`
- `__stdcall=__attribute__((__stdcall__))`

If you are running Polyspace on the command line in a UNIX shell, add double quotes around the -D option. For instance, use:

```
"-D __cdecl=__attribute__((__cdecl__))"
```

Troubleshooting Project Creation from Visual Studio Build

If you are trying to import a Visual Studio 2010 or Visual Studio 2012 project and `polyspace-configure` does not work properly, do the following:

- 1 Stop the `MSBuild.exe` process.
- 2 Set the environment variable `MSBUILDDISABLENODEREUSE` to 1.
- 3 Specify `MSBuild.exe` with the `/nodereuse:false` option.
- 4 Restart the Polyspace configuration tool:

```
polyspace-configure.exe -lang cpp <MSVS path>/msbuild sample.sln
```

See Also

`polyspace-configure`

Error Processing Macro with Semicolon in Build System

Issue

You see this error when creating a Polyspace project or options file from your build system:

```
Could not process macro containing a semicolon
```

Cause

Some options in your build system use semicolons in the replacement list of a macro. Automatic project creation from build systems does not support this usage. For instance, a macro `OK` with this replacement list can cause issues:

```
{printf("OK");flush();}
```

The use of semicolons in replacement lists is not supported because a Polyspace project or options file created from your build system itself uses semicolon separators to separate macro definitions. For details on the Polyspace options that define macros, see:

- `Preprocessor definitions (-D)`: This option defines macros.
- `-options-for-sources`: This option collects several macro definitions, separated by semicolon.

Solution

Define the macro in a header file instead of in the build system. For instance, define the macro `OK` like this in a header file:

```
#ifndef OK_DEFINED
#define OK_DEFINED
#define OK {printf("OK");flush();}
#endif
```

Provide the header file only for the purposes of Polyspace analysis using the option `Include (-include)`.

Could Not Find Include File

Issue

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.

Cause

By default, Polyspace 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.

Solution

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 7-12.

- 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 7-23.

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 7-12
- “Run Polyspace on AUTOSAR Code Using Build Command” on page 7-23
- “Troubleshoot Polyspace Analysis of AUTOSAR Code” on page 7-17

Conflicting Universal Unique Identifiers (UUIDs)

Issue

You see one or both of these errors when creating a Polyspace project from AUTOSAR XML and source 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.

Solution

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 7-12
- "Troubleshoot Polyspace Analysis of AUTOSAR Code" on page 7-17

Data Type Not Recognized

Issue

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.

Cause

When creating a Polyspace project, 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 you use a data type that is not in the Software Component specification, the analysis does not recognize this data type.

You can find the data types imported using the file `autosar_model_key_elements.html` in the AUTOSAR subfolder of your project folder. The file has data types in the `DataTypes` section in this format:

indirect	<code>pkg.types.app.Array_2_n320to320</code>
indirect	<code>pkg.types.app.Boolean</code>

The text `indirect` in the first column indicates that the data types are automatically imported.

Solution


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`.

The file `autosar_model_key_elements.html` shows data types that are explicitly imported using entries like this:

name	<code>tst003.typ.app.Boolean</code>
------	-------------------------------------

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 7-12
- “Troubleshoot Polyspace Analysis of AUTOSAR Code” on page 7-17

Undefined Identifier Error

Issue

Polyspace verification fails during the compilation phase with a message about undefined identifiers.

The message indicates that Polyspace cannot find a variable definition. Therefore, it cannot identify the variable type.

Possible Cause: Missing Files

The source code you provided does not contain the variable definition. For instance, the variable is defined in an include file that Polyspace cannot find.

If you `#include`-d the include file in your source code but did not add it to your Polyspace project, you see a previous warning:

```
Warning: could not find include file "my_include.h"
```

Solution

If the variable definition occurs in an include file, add the folder that contains the include file.

- In the user interface of the Polyspace desktop products, add the folder to your project.

For more information, see “Add Source Files for Analysis in Polyspace User Interface” on page 1-2.

- At the command line, use the flag `-I` with the `polyspace-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 9-24.

Possible Cause: Declaration Embedded in `#ifdef` Statements

The variable is declared in a branch of an `#ifdef macro_name` preprocessor directive. For instance, the declaration of a variable `max_power` occurs as follows:

```
#ifdef _WIN32
    #define max_power 31
#endif
```

Your compilation toolchain might consider the macro `macro_name` as implicitly defined and execute the `#ifdef` branch. However, the Polyspace compilation might not consider the macro as defined. Therefore, the `#ifdef` branch is not executed and the variable `max_power` is not declared.

Solution

To work around the compilation error, do one of the following:

- Use **Target & Compiler** options to directly specify your compiler. For instance, to emulate a Visual C++ compiler, set the **Compiler** to `visual12.0`. See “Target and Compiler”.
- Define the macro explicitly using the option `Preprocessor definitions (-D)`.

Note If you create a Polyspace by tracing your build commands, most **Target & Compiler** options are automatically set.

Possible Cause: Project Created from Non-Debug Build

This can be a possible cause only if the undefined identifier occurs in an `assert` statement (or an equivalent Visual C++ macro such as `ASSERT` or `VERIFY`).

Typically, you come across this error in the following way. You create a Polyspace project from a build system in non-debug mode. When you run an analysis on the project, you face a compilation error from an undefined identifier in an `assert` statement. You find that the identifier `my_identifier` is defined in a `#ifndef NDEBUG` statement, for instance as follows:

```
#ifndef NDEBUG
int my_identifier;
#endif
```

The C standard states that when the `NDEBUG` macro is defined, all `assert` statements must be disabled.

Most IDEs define the `NDEBUG` macro in their build systems. When you build your source code in your IDE in non-debug mode, code in a `#ifndef NDEBUG` statement is removed during preprocessing. For instance, in the preceding example, `my_identifier` is not defined. If `my_identifier` occurs only in `assert` statements, it is not used either, because `NDEBUG` disables `assert` statements. You do not have compilation errors from undefined identifiers and your build system executes successfully.

Polyspace does not disable `assert` statements even if `NDEBUG` macro is defined because the software uses `assert` statements internally to enhance verification.

When you create a Polyspace project from your build system, if your build system defines the `NDEBUG` macro, it is also defined for your Polyspace project. Polyspace removes code in a `#ifndef NDEBUG` statement during preprocessing, but does not disable `assert` statements. If `assert` statements in your code rely on the code in a `#ifndef NDEBUG` statement, compilation errors can occur.

In the preceding example:

- The definition of `my_identifier` is removed during preprocessing.
- `assert` statements are not disabled. When `my_identifier` is used in an `assert` statement, you get an error because of undefined identifier `my_identifier`.

Solution

To work around this issue, create a Polyspace project from your build system in debug mode. When you execute your build system in debug mode, `NDEBUG` is not defined. When you create a Polyspace project from this build, `NDEBUG` is not defined for your Polyspace project.

Depending on your project settings, use the option that enables building in debug mode. For instance, if your build system is gcc-based, you can define the `DEBUG` macro and undefine `NDEBUG`:

```
gcc -DDEBUG=1 -UNDEBUG *.c
```

Alternatively, you can disable the `assert` statements in your preprocessed code using the option `Preprocessor definitions (-D)`. However, Polyspace will not be able to emulate the `assert` statements.

Unknown Function Prototype Error

Issue

During the compilation phase, the software displays a warning or error message about unknown function prototype.

```
the prototype for function 'myfunc' is unknown
```

The message indicates that Polyspace cannot find a function prototype. Therefore, it cannot identify the data types of the function argument and return value, and has to infer them from the calls to the function.

To determine the data types for such functions, Polyspace follows the C99 Standard (ISO/IEC 9899:1999, Chapter 6.5.2.2: Function calls).

- The return type is assumed to be `int`.
- The number and type of arguments are determined by the first call to the function. For instance, if the function takes one `double` argument in the first call, for subsequent calls, the software assumes that it takes one `double` argument. If you pass an `int` argument in a subsequent call, a conversion from `int` to `double` takes place.

During the linking phase, if a mismatch occurs between the number or type of arguments or the return type in different compilation units, the verification stops. For more information, see “Conflicting Declarations in Different Translation Units” on page 22-60.

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 User Interface” on page 1-2.

- At the command line, use the flag `-I` with the `polyspace-code-prover` command.

For more information, see `-I`.

Error Related to #error Directive

Issue

The analysis stops with a message containing a `#error` directive. For instance, the following message appears: `#error directive: !Unsupported platform; stopping!`.

Cause

You typically use the `#error` directive in your code to trigger a fatal error in case certain macros are not defined. Your compiler implicitly defines the macros, therefore the error is not triggered when you compile your code. However, the default Polyspace compilation does not consider the macros as defined, therefore, the error occurs.

For instance, in the following example, the `#error` directive is reached only if the macros `__BORLANDC__`, `__VISUALC32__` or `__GNUC__` are not defined. If you use a GNU C compiler, for instance, the compiler considers the macro `__GNUC__` as defined and the error does not occur. However, if you use the default Polyspace compilation, it does not consider the macros as defined.

```
#if defined(__BORLANDC__) || defined(__VISUALC32__)
#define MYINT int
#elif defined(__GNUC__)
#define MYINT long
#else
#error !Unsupported platform; stopping!
#endif
```

Solution

For successful compilation, do one of the following:

- Specify a compiler such as `visual12.0` or `gnu4.9`. Specifying a compiler defines some of the compilation flags for the analysis.

For more information, see `Compiler (-compiler)`.

- If the available compiler options do not match your compiler, explicitly define one of the compilation flags `__BORLANDC__`, `__VISUALC32__`, or `__GNUC__`.

For more information, see `Preprocessor definitions (-D)`.

Large Object Error

Issue

The analysis stops during compilation with a message indicating that an object is too large.

Cause

The error happens when the software detects an object such as an array, union, structure, or class, that is too big for the pointer size of the selected target.

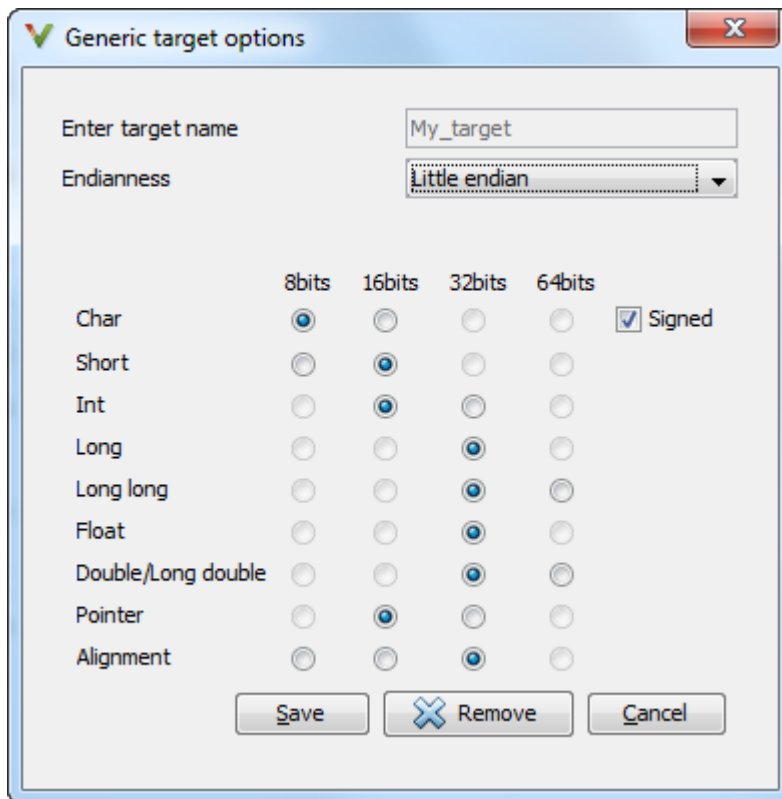
For instance, you get the message, `Limitation: struct or union is too large` in the following example. You specify a pointer size of 16 bits. The maximum object size allocated to a pointer, and therefore the maximum allowed size for an object, can be $2^{16}-1$ bytes. However, you declare a structure as follows:

- ```
struct S
{
 char tab[65536];
}s;
```
- ```
struct S
{
    char tab[65534];
    int val;
}s;
```

Solution

- 1 Check the pointer size that you specified through your target processor type. For more information, see `Target processor type (-target)`.

For instance, in the following, the pointer size for a custom target `My_target` is 16 bits.



- 2 Change your code or specify a different pointer size.

For instance, you can:

- Declare an array of smaller size in the structure.

If you are using a predefined target processor type, the pointer size is likely to be the same as the pointer size on your target architecture. Therefore, your declaration might cause errors on your target architecture.

- Change the pointer size of the target processor type that you specified, if possible.

Otherwise, specify another target processor type with larger pointer size or define your own target processor type. For more information on defining your own processor type, see `Generic target options`.

Note Polyspace imposes an internal limit of 128 MB on the size of data structures. Even if your target processor type specification allows data structures of larger size, this internal limit constrains the data structure sizes.

Errors Related to Generic Compiler

If you use a generic compiler, you can encounter this issue. For more information, see `Compiler (-compiler)`.

Issue

The analysis stops with an error message related to a non-ANSI C keyword, for instance, `data` or attributes such as `__attribute__((weak))`.

Depending on the location of the keyword, the error message can vary. For instance, this line causes the error message: `expected a ";"`.

```
data int tab[10];
```

Cause

The generic Polyspace compiler supports only ANSI C keywords. If you use a language extension, the generic compiler does not recognize it and treats the keyword as a regular identifier.

Solution

Specify your compiler by using the option `Compiler (-compiler)`.

If your compiler is not directly supported or is not based on a supported compiler, you can use the generic compiler. To work around the compilation errors:

- If the keyword is related to memory modelling, remove it from the preprocessed code. For instance, to remove the `data` keyword, enter `data=` for the option `Preprocessor definitions (-D)`.
- If the keyword is related to an attribute, remove attributes from the preprocessed code. Enter `__attribute__(x)=` for the option `Preprocessor definitions (-D)`.

If your code has this line:

```
void __attribute__((weak)) func(void);
```

And you remove attributes, the analysis reads the line as:

```
void func(void);
```

When you use these workarounds, your source code is not altered.

Errors Related to Keil or IAR Compiler

If you use the compiler, Keil or IAR, you can encounter this issue. For more information, see `Compiler (-compiler)`.

Missing Identifiers

Issue

The analysis stops with the error message, `expected an identifier`, as if an identifier is missing. However, in your source code, you can see the identifier.

Cause

If you select Keil or IAR as your compiler, the software removes certain keywords during preprocessing. If you use these keywords as identifiers such as variable names, a compilation error occurs.

For a list of keywords that are removed, see “Supported Keil or IAR Language Extensions” on page 9-19.

Solution

Specify that Polyspace must not remove the keywords during preprocessing. Define the macros `__PST_KEIL_NO_KEYWORDS__` or `__PST_IAR_NO_KEYWORDS__`.

For more information, see `Preprocessor definitions (-D)`.

Errors Related to Diab Compiler

If you choose `diab` for the option `Compiler` (`-compiler`), you can encounter this issue.

Issue

During Polyspace analysis, you see an error related to a keyword specific to the Diab compiler. For instance, you see an error related to the `restrict` keyword.

Cause

You typically use a compiler flag to enable the keyword. The Polyspace analysis does not enable these keywords by default. You have to make Polyspace aware of your compiler flags.

The Polyspace analysis does not enable these keywords by default to prevent compilation errors. Another user might not enable the keyword and instead use the keyword name as a regular identifier. If Polyspace treats the identifier as a keyword, a compilation error will occur.

Solution

Use the command-line option `-compiler-parameter` in your Polyspace analysis as follows. You use this command-line option to make Polyspace aware of your compiler flags. In the user interface of the Polyspace desktop products, you can enter the command-line option in the field `Other`. You can enter the option multiple times.

The argument of `-compiler-parameter` depends on the keyword that causes the error. Once you enable the keyword, do not use the keyword name as a regular identifier. For instance, once you enable the keyword `pixel`, do not use `pixel` as a variable name. The statement `int pixel = 1` causes a compilation error.

- `restrict` keyword:

You typically use the compiler flag `-Xlibc-new` or `-Xc-new`. For your Polyspace analysis, use `-compiler-parameter -Xc-new`

The following code will not compile with Polyspace unless you specify the compiler flag.

```
int sscanf(const char *restrict, const char *restrict, ...);
```

- PowerPC AltiVec vector extensions such as the `vector` type qualifier:

You typically use the compiler flag `-tPPCALLAV:.` For your Polyspace analysis, use `-compiler-parameter -tPPCALLAV:`

The following code will not compile with Polyspace unless you specify the compiler flag.

```
vector unsigned char vbyte;
vector bool vbool;
vector pixel vpx;

int main(int argc, char** argv)
{
```

```
    return 0;
}
```

- Extended keywords such as `pascal`, `inline`, `packed`, `interrupt`, `extended`, `__X`, `__Y`, `vector`, `pixel`, `bool` and others:

You typically use the compiler flag `-Xkeywords=`. For your Polyspace analysis, use
`-compiler-parameter -Xkeywords=0xFFFFFFFF`

The following code will not compile with Polyspace unless you specify the compiler flag.

```
packed(4) struct s2_t {
    char b;
    int i;
} s2;

packed(4,2) struct s3_t {
    char b;
} s3;

int pascal foo = 4;

int main(int argc, char** argv) {
    foo++;
    return 0;
}
```

Errors Related to Green Hills Compiler

If you choose `greenhills` for the option `Compiler` (`-compiler`), you encounter this issue.

Issue

During Polyspace analysis, you see an error related to vector data types specific to Green Hills target `rh850`. For instance, you see an error related to identifier `__ev64_u16__`.

Cause

When compiling code using the Green Hills compiler with target `rh850`, to enable single instruction multiple data (SIMD) vector instructions, you specify two flags:

- `-rh850_simd`: You enable intrinsic functions that support SIMD vector instructions. The functions are defined in your compiler header files. These data types are available:
 - `__ev64_u16__`
 - `__ev64_s16__`
 - `__ev64_u32__`
 - `__ev64_s32__`
 - `__ev64_u64__`
 - `__ev64_s64__`
 - `__ev64_opaque__`
 - `__ev128_opaque__`
- `-rh850_fpsimd`: You enable intrinsic functions that support floating-point SIMD vector instructions. The functions are defined in your compiler header files. These data types are available:
 - `__ev128_f32__`
 - `__ev256_f32__`

The Polyspace analysis does not enable SIMD support by default. You must identify your compiler flags to Polyspace.

Solution

In your Polyspace analysis, use the command-line option `-compiler-parameter`. In the user interface, you can enter the command-line option in the `Other` field, under the **Advanced Settings** in the **Configuration** pane.

- `-rh850_simd`: For your Polyspace analysis, use
`-compiler-parameter -rh850_simd`
- `-rh850_fpsimd`: For your Polyspace analysis, use
`-compiler-parameter -rh850_fpsimd`

Note

- `__ev128_opaque__` is 16 bytes aligned in Polyspace.
 - `__ev256_f32__` is 32 bytes aligned in Polyspace.
-

Errors Related to TASKING Compiler

If you choose tasking for the option Compiler (-compiler), you can encounter this issue.

Issue

During Polyspace analysis, you see an error related to a Special Function Register data type.

Cause

When compiling with the TASKING compiler, you typically use the following compiler flags to specify where Special Function Register (SFR) data types are declared:

- `--cpu=xxx`: The compiler implicitly `#includes` the file `sfr/regxxx.sfr` in your source files. Once `#include`-ed, you can use Special Function Registers (SFR-s) declared in that `.sfr` file.
- `--alternative-sfr-file`: The compiler uses an alternative SFR file instead of the regular SFR file. You can use Special Function Registers (SFR-s) declared in that alternative SFR file.

If you specify the TASKING compiler for your Polyspace analysis, the analysis makes the following assumptions about these compiler flags:

- `--cpu=xxx`: The analysis chooses a specific value of `xxx`. If you use a different value with your TASKING compiler, you can encounter an error during Polyspace analysis.

The `xxx` value that the Polyspace analysis uses depends on your choice of Target processor type (-target):

- `tricore: tc1793b`
- `c166: xc167ci`
- `rh850: r7f701603`
- `arm: ARMv7M`
- `--alternative-sfr-file`: The analysis assumes that you do not use an alternative SFR file. If you use one, you can encounter an error.

Solution

Use the command-line option `-compiler-parameter` in your Polyspace analysis as follows. You use this command-line option to make Polyspace aware of your compiler flags. In the user interface, you can enter the command-line option in the field `Other`. You can enter the option multiple times.

- `--cpu=xxx`: For your Polyspace analysis, use
`-compiler-parameter --cpu=xxx`

Here, `xxx` is the value that you use when compiling with your compiler.

- `--alternative-sfr-file`: For your Polyspace analysis, use
`-compiler-parameter --alternative-sfr-file`

If you still encounter an error because Polyspace is not able to locate your `.asfr` file, explicitly `#include` your `.asfr` file in the preprocessed code using the option `Include` (-include).

Typically, the path to the file is *Tasking_C166_INSTALL_DIR*\include\sfr\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.

You can also encounter the same issue with alternative sfr files when you trace your build command. For more information, see “Requirements for Project Creation from Build Systems” on page 9-15.

Errors from In-Class Initialization (C++)

When a data member of a class is declared `static` in the class definition, it is a *static member* of the class. You must initialize static data members outside the class because they exist even when no instance of the class has been created.

```
class Test
{
public:

    static int m_number = 0;
};
```

Error message:

Error: a member with an in-class initializer must be const

Corrected code:

in file Test.h	in file Test.cpp
<pre>class Test { public: static int m_number; };</pre>	<pre>int Test::m_number = 0;</pre>

Errors from Double Declarations of Standard Template Library Functions (C++)

Consider the following code.

```
#include <list>

void f(const std::list<int*>::const_iterator it) {}
void f(const std::list<int*>::iterator it) {}
void g(const std::list<int*>::const_reverse_iterator it) {}
void g(const std::list<int*>::reverse_iterator it) {}
```

The declared functions belong to `list` container classes with different iterators. However, the software generates the following compilation errors:

```
error: function "f" has already been defined
error: function "g" has already been defined
```

You would also see the same error if, instead of `list`, the specified container was `vector`, `set`, `map`, or `deque`.

To avoid the double declaration errors, do one of the following:

- Deactivate automatic stubbing of standard template library functions. For more information, see `No STL stubs (-no-stl-stubs)`.
- Define the following Polyspace preprocessing directives:
 - `__PST_STL_LIST_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_VECTOR_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_SET_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_MAP_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_DEQUE_CONST_ITERATOR_DIFFER_ITERATOR__`

For example, for the given code, run analysis at the command line with the following flag. The flag defines the appropriate directive for the `list` container.

```
-D __PST_STL_LIST_CONST_ITERATOR_DIFFER_ITERATOR__
```

For more information on defining preprocessor directives, see `Preprocessor definitions (-D)`.

Errors Related to GNU Compiler

If you choose `gnu` for the option `Compiler` (`-compiler`), you can encounter this issue.

Issue

The Polyspace analysis stops with a compilation error.

Cause

You are using certain advanced compiler-specific extensions that Polyspace does not support. See “Limitations”.

Solution

For easier portability of your code, avoid using the extensions.

If you want to use the extensions and still analyze your code, wrap the unsupported extensions in a preprocessor directive. For instance:

```
#ifdef POLYSPACE
    // Supported syntax
#else
    // Unsupported syntax
#endif
```

For regular compilation, do not define the macro `POLYSPACE`. For Polyspace analysis, enter `POLYSPACE` for the option `Preprocessor definitions` (`-D`).

If the compilation error is related to assembly language code, use the option `-asm-begin` `-asm-end`.

Errors Related to Visual Compilers

The following messages appear if the compiler is based on a Visual compiler. For more information, see `Compiler (-compiler)`.

Import Folder

When a Visual application uses `#import` directives, the Visual C++ compiler generates a header file with extension `.tlh` that contains some definitions. To avoid compilation errors during Polyspace analysis, you must specify the folder containing those files.

Original code:

```
#include "stdafx.h"
#include <comdef.h>
#import <MsXml.tlb>
MSXML::_xml_error e ;
MSXML::DOMDocument* doc ;
int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

Error message:

```
"../sources/ImportDir.cpp", line 7: catastrophic error: could not
open source file "../MsXml.tlh"
    #import <MsXml.tlb>
```

The Visual C++ compiler generates these files in its “build-in” folder (usually Debug or Release). In order to provide those files:

- Build your Visual C++ application.
- Specify your build folder for the Polyspace analysis.

pragma Pack

Using a different value with the compile flag (`#pragma pack`) can lead to a linking error message.

Original code:

test1.cpp	type.h	test2.cpp
<pre>#pragma pack(4) #include "type.h"</pre>	<pre>struct A { char c ; int i ; } ;</pre>	<pre>#pragma pack(2) #include "type.h"</pre>

Error message:

```
Pre-linking C++ sources ...
"../sources/type.h", line 2: error: declaration of class "A" had
a different meaning during compilation of "test1.cpp"
(class types do not match)
    struct A
```

```
^
  detected during compilation of secondary translation unit
"test2.cpp"
```

To continue the analysis, use the option `Ignore pragma pack directives (-ignore-pragma-pack)`.

C++/CLI

Polyspace does not support Microsoft C++/CLI, a set of language extensions for .NET programming.

You can get errors such as:

```
error: name must be a namespace name
|         using namespace System;
```

Or:

```
error: expected a declaration
|         public ref class Form1 : public System::Windows::Forms::Form
```

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.

<code>file1.c</code>	<code>file2.c</code>
<pre>extern int x; void main(void) { /* Variable x used */ }</pre>	<pre>volatile int x;</pre>

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`.

<code>file1.c</code>	<code>file2.c</code>
<pre>int input(void); void main() { int val = input(); }</pre>	<pre>float input(void) { float x = 1.0; return x; }</pre>

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 2-4.

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`.

<code>file1.cpp</code>	<code>file2.cpp</code>
<pre>#include "field.h" int main() { a_struct s; init_a_struct(&s); return 0; }</pre>	<pre>#define DEBUG #include <string.h> #include "field.h" void init_a_struct(a_struct* s) { memset(s, 0, sizeof(*s)); }</pre>
<p>field.h:</p> <pre>struct a_struct { int n; #ifdef DEBUG int debug; #endif };</pre>	

When you open the preprocessed files `file1.ci` and `file2.ci`, you see the conflicting declarations.

<code>file1.ci</code>	<code>file2.ci</code>
<pre>struct a_struct { int n; };</pre>	<pre>struct a_struct { int n; int debug; };</pre>

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`.

file1.cpp	file2.cpp
<pre>#include "bool.h" int main() { return 0; }</pre>	<pre>#define false 0 #define true (!false) #include "bool.h"</pre>
<p>bool.h:</p> <pre>template <class T> struct a_struct { bool flag; T t; a_struct() { flag = true; } };</pre>	

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 “#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`.

file1.cpp	file2.cpp
<pre>int main() { return 0; }</pre>	<pre>#pragma pack(1) #include "pack.h"</pre>
<p>pack.h:</p> <pre>struct a_struct { char ch; short sh; };</pre>	

Solution

Enter the `#pragma pack(n)` statement in the header file so that it applies to all source files that include the header.

Errors from Conflicts with Polyspace Header Files

Issue

You see compilation errors from header files included by Polyspace.

For instance, the error message refers to one of the subfolders of *polyspaceroot*\polyspace\verifier\cxx\include.

Typically, the error message is related to a standard library function.

Cause

If your compiler defines a standard library function or another construct and you do not provide the path to your compiler header files, Polyspace uses its own implementation of the function.

If your compiler definitions differ from the corresponding Polyspace definitions, the verification stops with an error.

Solution

Specify the folder containing your compiler header files.

- In the user interface, add the folder to your project.

For more information, see “Add Source Files for Analysis in Polyspace User Interface” on page 1-2.

- At the command line, use the flag `-I` with the `polyspace-code-prover` command.

For more information, see `-I`.

For compilation with GNU C on UNIX-based platforms, use `/usr/include`. On embedded compilers, the header files are typically in a subfolder of the compiler installation folder. Examples of include folders are given for some compilers.

- Wind River Diab: For instance, `/apps/WindRiver/Diab/5.9.4/diab/5.9.4.8/include/`.
- IAR Embedded Workbench: For instance, `C:\Program Files\IAR Systems\Embedded Workbench 7.5\arm\inc`.
- Microsoft Visual Studio: For instance, `C:\Program Files\Microsoft Visual Studio 14.0\VC\include`.

Consult your compiler documentation for the path to your compiler header files. Alternatively, see “Provide Standard Library Headers for Polyspace Analysis” on page 9-14.

C++ Standard Template Library Stubbing Errors

Issue

The analysis stops with an error message that refers to class templates such as `map` and `vector` from the Standard Template Library.

Often, the error message states that either an operator cannot be found or more than one operator matches the given operands.

Cause

Polyspace software provides an efficient implementation of all class templates from the Standard Template Library (STL). If your source code redeclares the templates, the analysis can stop with an error message.

Solution

To use your own implementations of templates from the Standard Template Library:

- 1 Disable the Polyspace implementations using the option `No STL stubs (-no-stl-stubs)`.
- 2 Add the folders containing your implementations to the verification.

- In the user interface, add the folder to your project.

For more information, see “Add Source Files for Analysis in Polyspace User Interface” on page 1-2.

- At the command line, use the flag `-I` with the `polyspace-code-prover` command.

For more information, see `-I`.

Note Using your own template definitions can cause other compilation and linking errors.

Lib C Stubbing Errors

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.

Errors from Using Namespace std Without Prefix

Issue

The Polyspace analysis stops with an error message such as:

```
error: the global scope has no "modfl"
```

The line highlighted in the error uses a function from the standard library without the `std::` prefix.

Cause

Some compilers allow using members of the standard library namespace without explicitly specifying the `std::` prefix. For such compilers, your code can contain lines like this:

```
using ::mblen;
```

where `mblen` is a member of the C++ standard library. Polyspace compilation considers the members as part of the global namespace and shows an error.

Solution

It is a good practice to qualify members of the standard library with the `std::` prefix. For instance, to use the `mblen` function in the preceding example, rewrite the line as:

```
using std::mblen;
```

To continue to retain the current code and work around the Polyspace error, use the analysis option `-using-std`. If you are running the analysis in the Polyspace user interface, enter the option in the **Other** field. See **Other**.

Errors from Assertion or Memory Allocation Functions

Issue

Polyspace uses its own implementation of standard library functions for more efficient analysis. If you redefine a standard library function and provide the function body to Polyspace, the analysis uses your definition.

However, for certain standard library functions, Polyspace continues to use its own implementations, even if you redefine the function and provide the function body. The functions include `assert` and memory allocation functions such as `malloc`, `calloc` and `alloca`.

You see a warning message like the following:

```
Body of routine "malloc" was discarded.
```

Cause

These functions have special meaning for the Polyspace analysis, so you are not allowed to redefine them. For instance:

- The Polyspace implementation of the `malloc` function allows the software to check if memory allocated using `malloc` is freed later.
- The Polyspace implementation of `assert` is used internally to enhance analysis.

Solution

Unless you particularly want your own redefinitions to be used, ignore the warning. The analysis results are based on Polyspace implementations of the standard library function, which follow the original function specifications.

If you want your own redefinitions to be used and you are sure that your redefined function behaves the same as the original function, rename the functions. You can rename the function only for the purposes of analysis using the option `Preprocessor definitions (-D)`. For instance, to rename a function `malloc` to `my_malloc`, use `malloc=my_malloc` for the option argument.

Eclipse Java Version Incompatible with Polyspace Plug-in

In this section...

"Issue" on page 22-71

"Cause" on page 22-71

"Solution" on page 22-71

Issue

After installing the Polyspace plug-in for Eclipse, when you open the Eclipse or Eclipse-based IDE, you see this error message:

```
Java 7 required, but the current java version is 1.6.
You must install Java 7 before using Polyspace plug in.
```

You see this message even if you install Java 7 or higher.

Cause

Despite installing Java 7 or higher, the Eclipse or Eclipse-based IDE still uses an older version.

Solution

Make sure that the Eclipse or Eclipse-based IDE uses the compatible Java version.

- 1 Open the *executable_name.ini* file that occurs in the root of your Eclipse installation folder.

If you are running Eclipse, the file is *eclipse.ini*.

- 2 In the file, just before the line *-vmargs*, enter:

```
-vm
java_install\bin\javaw.exe
```

Here, *java_install* is the Java installation folder.

For instance, your product installation comes with the required Java version for certain platforms. You can force the Eclipse or Eclipse-based IDE to use this version. In your *.ini* file, enter the following just before the line *-vmargs*:

```
-vm
polyspaceroot\sys\java\jre\arch\jre\bin\javaw.exe
```

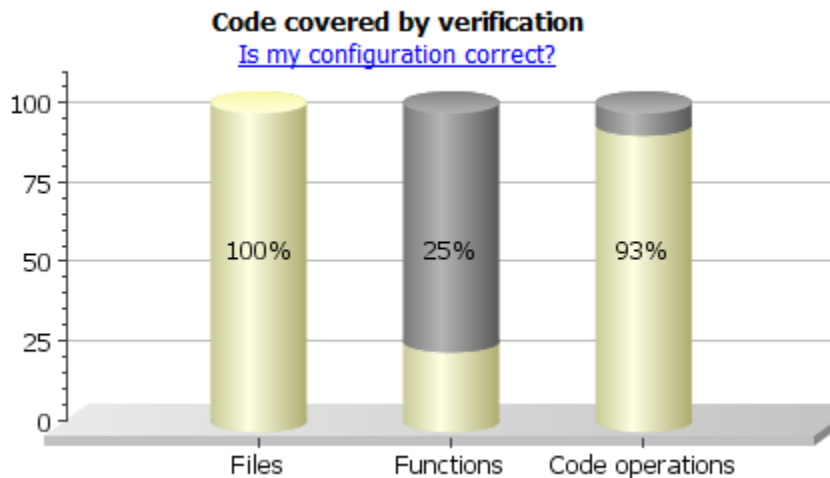
Here, *polyspaceroot* is your product installation folder, for instance, *C:\Program Files\Polyspace\R2019a* and *arch* is *win32* or *win64* depending on the product platform. Note that *-vm* and the path to *javaw.exe* must be on separate lines.

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:

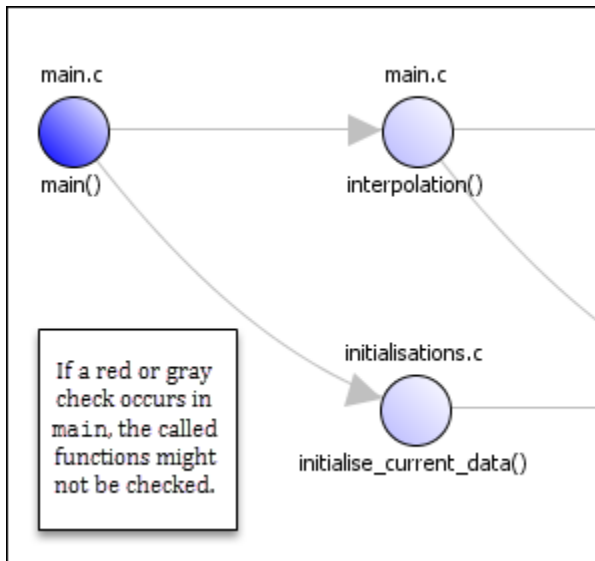
- How the Polyspace compilation works, see “Troubleshoot Compilation and Linking Errors” on page 22-6.
- Specific compilation errors, see “Troubleshoot Compilation Errors”.

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” on page 16-33.

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.

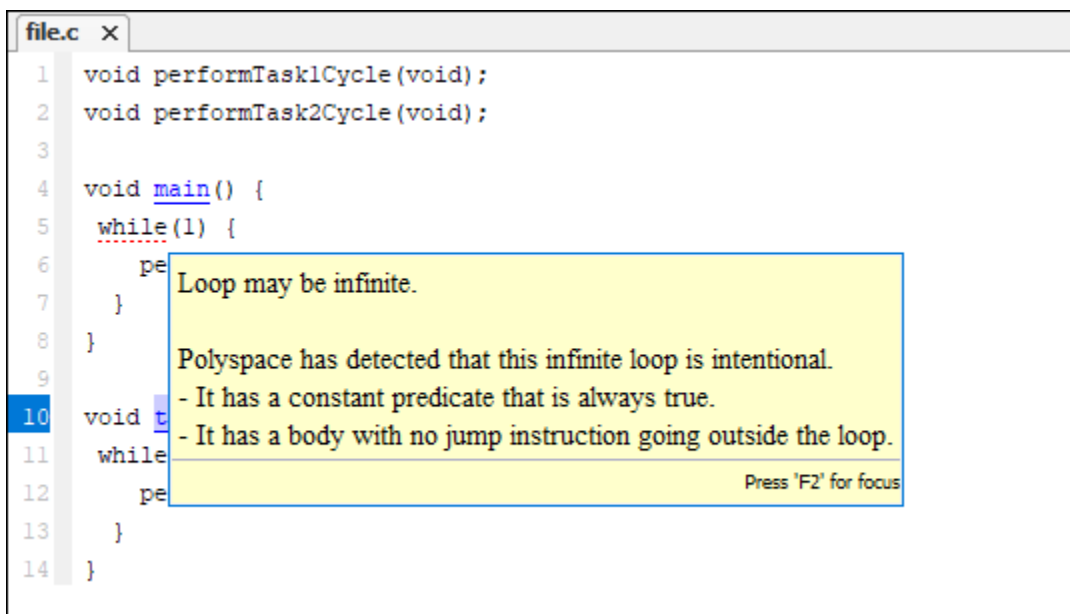
For instance, in this example, `task2` is not checked even if you specify it as an entry point. The reason is the infinite loop in the `main` function.

```
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 11-16.

Source Files or Functions Not Displayed in Results List

In this section...
“Issue” on page 22-78
“Possible Cause: Files Not Verified” on page 22-78
“Possible Cause: Filters Applied” on page 22-79

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” on page 16-16.

Solution

Polyspace does not verify a source file or function when one of the following situations occur.

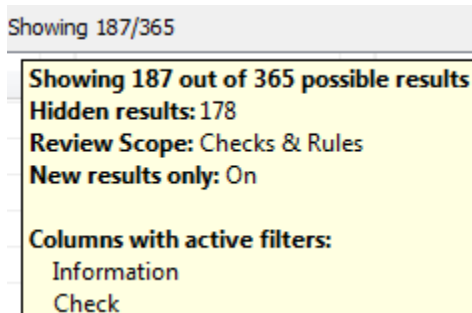
Situation	Fix
<p>The file or function does not contain an operation on which a check is required.</p> <p>For instance, a function contains calls to other functions only. If none of the called functions contains an error that lead to a Non-terminating call error in the calling function, the calling function does not contain a check.</p>	No fix required.
<p>All functions in the source file are not called, are called from unreachable code or are called following red checks.</p> <p>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.</p>	<p>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).</p>

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 11-16.
<p>If your source files do not contain a <code>main</code> function, Polyspace generates a <code>main</code> function. The generated <code>main</code> calls the functions that you specify using certain analysis options.</p> <p>If your analysis options are such that the generated <code>main</code> does not call all the functions in a source file, Polyspace does not verify the source file.</p>	<p>See if you have to change the main generation options associated with your verification.</p> <p>For more information on the options, see:</p> <ul style="list-style-type: none"> • 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” on page 19-2.

Coding Standard Violations Not Displayed

Issue

You expect a coding rule violation on a line of code but the Polyspace analysis does not show the violation.

Possible Cause: Rule Checker Not Enabled

You might be looking for a reduced subset of coding rules.

For instance, if you check for MISRA C: 2012 rules, by default, the analysis looks for the mandatory-required subset only.

Solution

Check the coding rules options that you use. See:

- Check MISRA C:2004 (-misra2)
- Check MISRA C:2012 (-misra3)
- Check MISRA C++:2008 (-misra-cpp)
- Check JSF AV C++ rules (-jsf-coding-rules)

Possible Cause: Rule Violations in Header Files

All coding rule violations in the file might be suppressed.


For instance, by default, coding rule violations are suppressed from header files that are not in the same location as the source files.


Solution

Check the files where you suppress coding rule violations. See `Do not generate results for (-do-not-generate-results-for)`.

Possible Cause: Rule Violations in Macros

The rule violation occurs in a macro expansion. To save you from reviewing the same violation multiple times, the violation is shown on the macro definition instead of the macro usage. If the definition occurs in a header file, it might be suppressed from the results.

On the **Source** pane, you can tell if a line contains a macro expansion. Look for the  icon.

```
110  s8_ret = (s8) NA_VALUE;
```

Solution

Find the macro definition and see if it occurs in a header file. Determine if you are suppressing coding rule violations from header files. See `Do not generate results for (-do-not-generate-results-for)`.

Possible Cause: Rule Violations in Sources Because of Header Files

For specific rules, a rule violation might occur because of the content of a header file. If you suppress that header file from analysis, the rule violation disappears.

For instance, MISRA C:2012 Rule 1.1 is violated if too many macros are defined in a file. Sometimes, the macros might not be defined in the source file itself but in header files included in the source file. In this case, the rule violation appears on the `#include` statement that includes the offending header file. If you suppress this header file using the option `Do not generate results for (-do-not-generate-results-for)`, the violation disappears.

Solution

Unsuppress header files from coding rules checking and see if the rule violation appears on a `#include` statement. This indicates that the violation occurs only if the header file is taken into account.

Possible Cause: Compilation Errors

If any source file in the analysis does not compile, coding rules checking will be incomplete. The coding rules checker results:

- May not contain full results for files that did not compile
- May not contain full results for the files that did compile as some rules are checked only after compilation is complete

Check for compilation errors. See “View Error Information When Analysis Stops” on page 22-3.

Note When you enable the Compilation Assistant *and* coding rules checking, the software does not report coding rule violations if there are compilation errors.

Possible Cause: Code Prover Analysis with Lower Verification Level

If you run a Code Prover analysis to source compliance checking using the option `Verification level (-to)`, you might not see violations of some rules. These rules are checked in the later stages of a Code Prover analysis.

This reasoning applies to specific rules and does not apply to a Bug Finder analysis. See “Check for Coding Standard Violations” on page 12-2.

Solution

If you do not see a violation of one of those rules, check if your Code Prover analysis runs up to source compliance checking only. Use a higher value for the option `Verification level (-to)`.

Incorrect Behavior of 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 User Interface" on page 1-2.

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.

Insufficient Memory During Report Generation

Message

```
....  
Exporting views...  
Initializing...  
Polyspace Report Generator  
Generating Report  
.....  
    Converting report  
Opening log file: C:\Users\user\AppData\Local\Temp\java.log.7512  
Document conversion failed  
.....  
Java exception occurred:  
java.lang.OutOfMemoryError: Java heap space
```

Possible Cause

During generation of very large reports, the software can sometimes indicate that there is insufficient memory.

Solution

If this error occurs, try increasing the Java heap size. The default heap size in a 64-bit architecture is 1024 MB.

To increase the size:

- 1 Navigate to *polyspaceroot*\polyspace\bin*architecture*. Where:
 - *polyspaceroot* is the installation folder.
 - *architecture* is your computer architecture, for instance, win32, win64, etc.
- 2 Change the default heap size that is specified in the file, *java.opts*. For example, to increase the heap size to 2 GB, replace 1024m with 2048m.
- 3 If you do not have write permission for the file, copy the file to another location. After you have made your changes, copy the file back to *polyspaceroot*\polyspace\bin*architecture*.

Errors with Temporary Files

Polyspace produces some temporary files during analysis. The following issues are related to storage of temporary files.

No Access Rights

When running verification, you get an error message that Polyspace could not create a folder for writing temporary files. For instance, the error message can be as follows:

```
Unable to create folder "C:\Temp\Polyspace\foldername
```

Cause

Polyspace produces some temporary files during analysis. If you do not have write permissions for the folder used to store the files, you can encounter the error.

Solution

There are two possible solutions to this error:

- Change the permissions of your temporary folder so you have full read and write privileges.

To learn how Polyspace determines the temporary folder location, see “Storage of Temporary Files” on page 1-12.

- Use the option `-tmp-dir-in-results-dir`. Instead of the standard temporary folder, Polyspace uses a subfolder of the results folder.

No Space Left on Device

When running verification, you get an error message that there is no space on a device.

Cause

If you do not have sufficient space on for the folder used to store the files, you can encounter the error.

Solution

There are two possible solutions to this error:

- Change the temporary folder to a drive that has enough disk space.

To learn how Polyspace determines the temporary folder location, see “Storage of Temporary Files” on page 1-12.

- Use the option `-tmp-dir-in-results-dir`. Instead of the standard temporary folder, Polyspace uses a subfolder of the results folder.

Cannot Open Temporary File

When running verification, you get an error message that Polyspace could not open a temporary file.

Cause

You defined the path for storing temporary files by using the environment variable `RTE_TMP_DIR`. You either used a relative path for the temporary folder, the folder does not exist or you do not have access rights to the folder.

Solution

There are two possible solutions to this error:

- Instead of defining a temporary folder specific to Polyspace through `RTE_TMP_DIR`, use a standard temporary folder.

To learn how Polyspace determines the temporary folder location, see “Storage of Temporary Files” on page 1-12.

- 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.

Error or Slow Runs from Disk Defragmentation and Anti-virus Software

Issue

In some cases, anti-virus software checks can noticeably slow down a Polyspace analysis. This reduction occurs because the software checks the temporary files produced by the Polyspace analysis.

You see noticeably slow analysis for a simple project or the analysis stops with an error message like the following:

```
Some stats on aliases use:
  Number of alias writes:      22968
  Number of must-alias writes: 3090
  Number of alias reads:       0
  Number of invisibles:       949
Stats about alias writes:
  biggest sets of alias writes: foo1:a (733), foo2:x (728), foo1:b (728)
  procedures that write the biggest sets of aliases: foo1 (2679), foo2 (2266),
                                                    foo3 (1288)
**** C to intermediate language translation - 17 (P_PT) took 44real, 44u + 0s (1.4gc)
exception SysErr(OS.SysErr(name="Directory not empty", syserror=notempty)) raised.
unhandled exception: SysErr: No such file or directory [noent]
```

```
-----
---
--- Verifier has encountered an internal error.      ---
--- Please contact your technical support.          ---
---
-----
```

Possible Cause

A disk defragmentation tool or anti-virus software is running on your machine.

After starting an analysis, check the processes running and see if an anti-virus process is causing large amount of CPU usage (and possibly memory usage).

Solution

Try:

- Stopping the disk defragmentation tool.
- Deactivating the anti-virus software. Or, configuring exception rules for the anti-virus software to allow Polyspace to run without a failure.

For instance, you can try the following:

- Configure the anti-virus software to whitelist the Polyspace executables.

For instance, in Windows, with the anti-virus software Windows Defender, you can add an exclusion for the Polyspace installation folder C:\Program Files\Polyspace\R2019a, in particular, the .exe files in the subfolder polyspace\bin and the .exe files starting with ps_ in the subfolder bin\win64.

- Configure the anti-virus software to exclude your temporary folder, for example, C:\Temp, from the checking process.

See “Storage of Temporary Files” on page 1-12.

SQLite I/O Error

Issue

When you try to run Polyspace, you get this error message:

Cause

Polyspace uses an SQLite database for storing results. This error can appear when SQLite databases are saved on NFS (Network File System) folders.

Solution

Check the folder where you save Polyspace results. For instance, if you run Polyspace at the command line, check the option `-results-dir`.

If the folder is an NFS folder, use a local folder instead.

License Error -4,0

Issue

When you try to run Polyspace, you get this error message:

```
License Error -4,0
```

Possible Cause: Another Polyspace Instance Running

You can open multiple instances of Polyspace, but you can only run one code analysis at a time.

If you try to run Polyspace processes from multiple windows, you will get a License Error -4,0 error.

Solution

Only run one analysis at a time, including any command-line or plugin analyses.

Possible Cause: Prior Polyspace Run in Simulink or MATLAB Coder

If you run Polyspace on generated code in the Simulink user interface or in the MATLAB Coder app, you can get a license error if you try to run a subsequent analysis in the Polyspace user interface. You get the error even if the previous run is over.

Solution

Run the subsequent analysis using the method that you used before, that is, in the Simulink user interface or MATLAB Coder app.

If you want to run the analysis in the Polyspace user interface, close Simulink or MATLAB Coder and then rerun the analysis.