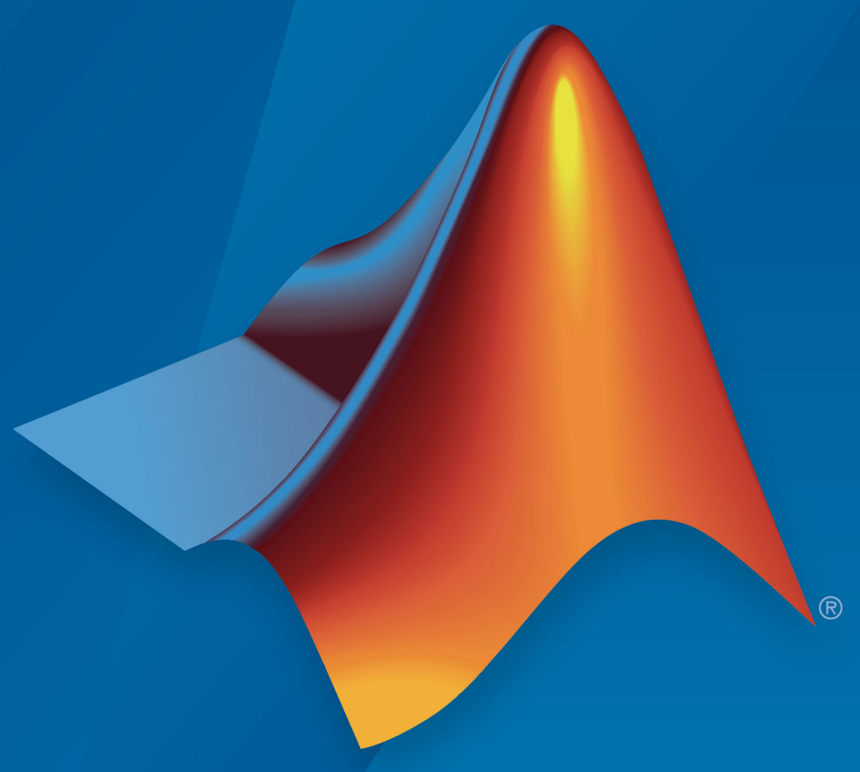
Polyspace® Code Prover™

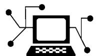
User's Guide



MATLAB® & SIMULINK®

R2017b

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.
3 Apple Hill Drive
Natick, MA 01760-2098

Polyspace® Code Prover™ User's Guide

© COPYRIGHT 2013–2017 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)

Contents

1 Introduction to Polyspace Products

Polyspace Verification **1-2**
Polyspace Verification **1-2**
Value of Polyspace Verification **1-2**

How Polyspace Verification Works **1-5**
What is Static Verification **1-5**
Exhaustiveness **1-6**

Related Products **1-7**
Polyspace Bug Finder **1-7**
Polyspace Products for Verifying Ada Code **1-7**
Tool Qualification and Certification **1-7**

2 How to Use Polyspace Software

Polyspace Verification and the Software Development Cycle **2-2**
Software Quality and Productivity **2-2**
Best Practices for Verification Workflow **2-3**

Implement Process for Verification **2-4**
Overview of the Polyspace Process **2-4**
Define Process to Meet Your Goals **2-4**
Apply Process to Assess Code Quality **2-5**
Improve Your Verification Process **2-5**

Sample Workflows for Polyspace Verification **2-6**
Overview of Verification Workflows **2-6**

Software Developers and Testers – Standard Development Process 2-6
Software Developers and Testers – Rigorous Development Process 2-9
Quality Engineers – Code Acceptance Criteria 2-12
Quality Engineers – Certification/Qualification 2-14
Model-Based Design Users — Verifying Generated Code . . . 2-15
Project Managers — Integrating Polyspace Verification with Configuration Management Tools 2-18

Define Your Requirements **2-19**
Define Broad Requirements for Verification 2-19
Define Specific Requirements for Verification 2-20

3 Setting up Project in User Interface

Create Project Automatically **3-2**

Requirements for Project Creation from Build Systems **3-5**

Compiler Not Supported for Project Creation from Build Systems **3-8**
Issue 3-8
Cause 3-8
Solution 3-8

Slow Build Process When Polyspace Traces the Build **3-16**
Issue 3-16
Cause 3-16
Solution 3-16

Check if Polyspace Supports Build Scripts **3-17**
Issue 3-17
Possible Cause 3-17
Solution 3-17

Troubleshooting Project Creation from MinGW Build **3-19**
Issue 3-19
Cause 3-19

Solution 3-19

Create Project Manually 3-20
Create Project 3-20
Specify Analysis Options 3-22

Create Project Using Configuration Template 3-25
Why Use Templates 3-25
Use Predefined Template 3-25
Create Your Own Template 3-26

Update Project 3-29
Change Folder Path 3-29
Refresh Source List 3-29
Refresh Project Created from Build Command 3-30
Add Source and Include Folders 3-30
Manage Include File Sequence 3-32
Change Analysis Options 3-32

Modularize Project Manually 3-34
Create New Module 3-34
Create Configurations in Module 3-35

Modularize Project Automatically 3-37

Create Project Using Visual Studio Information 3-41

Troubleshooting Project Creation from Visual Studio Build 3-43
Cannot Create Project from Visual Studio Build 3-43
Compilation Error After Creating Project from Visual Studio Build 3-43

4 Setting Up Polyspace User Interface

Organize Layout of Polyspace User Interface 4-2
Create Your Own Layout 4-2
Save and Reset Layout 4-3

Specify External Text Editor 4-4

Change Default Font Size 4-6

Customize Results Folder Location and Name 4-7

Storage of Polyspace Preferences 4-8

5 Emulating Your Runtime Environment

Specify Target Environment and Compiler Behavior 5-2
Extract Options from Build Command 5-3
Specify Options Explicitly 5-3

Provide Standard Library Headers for Polyspace Analysis 5-6

Language Extensions Supported by Default 5-8

Supported Keil or IAR Language Extensions 5-10
Special Function Register Data Type 5-10
Keywords Removed During Preprocessing 5-11

Supported C++ 2011 Language Extensions 5-12

Remove or Replace Keywords Before Compilation 5-15

Gather Compilation Options Efficiently 5-18

Verify C Application Without main Function 5-20
Generate main Function 5-20
Manually Write main Function 5-20

Verify C++ Classes 5-24
Verification of Classes 5-24
Methods and Class Specifics 5-26

Specify External Constraints 5-35
Create Constraint Template 5-35

Create Constraint Template After Analysis 5-36
Update Existing Template 5-37
Specify Constraints in Code 5-38

Constrain Global Variable Range **5-40**

Constrain Function Inputs **5-42**

Constrain Stubbed Functions **5-44**

Constraints **5-46**

XML File Format for Constraints **5-56**
Syntax Description — XML Elements 5-56
Valid Modes and Default Values 5-61

Provide Context for C Code Verification **5-65**
Control Variable Range 5-65
Control Function Call Sequence 5-65
Control Stubbing Behavior 5-66

Provide Context for C++ Code Verification **5-67**
Control Variable Range 5-67
Control Function Call Sequence 5-67

Verify Multitasking Applications **5-69**
Verify Multitasking Code with Automatic Concurrency Detection 5-69
Configure Multitasking Configuration Manually 5-70

Manually Model Tasks if main Contains Infinite Loop **5-72**
Run Multitasking Verification Without Modifying Code 5-72
Run Multitasking Verification After Modifying Code 5-73

Manually Model Scheduling of Tasks **5-76**
Specify Entry Points 5-77
Specify Definite Execution Sequence 5-77
Specify Indefinite Execution Sequence 5-78

Manually Protect Shared Variables from Concurrent Access **5-80**
Prerequisites 5-81
View Unprotected Access in Polyspace Results 5-81

Protect Shared Variables Using Temporally Exclusive Tasks . . . 5-82
Protect Shared Variables Using Critical Sections . . . 5-83

6 Running a Verification

Specify Results Folder . . . **6-2**
Create New Result Folder for Each Run . . . **6-2**
Change Parent Folder of Results Folders . . . **6-3**

Run Local Verification . . . **6-4**
Start Verification . . . **6-4**
Monitor Progress . . . **6-5**
Stop Verification . . . **6-5**
Open Results . . . **6-5**

Run Remote Verification . . . **6-7**
Start Verification . . . **6-7**
Monitor Progress . . . **6-8**
Stop Verification . . . **6-8**
Open Results . . . **6-8**

Phases of Verification . . . **6-10**

Run File-by-File Local Verification . . . **6-11**
Run Verification . . . **6-11**
Open Results . . . **6-11**

Run File-by-File Remote Verification . . . **6-14**
Run Verification . . . **6-14**
Open Results . . . **6-15**

Create Project Automatically at Command Line . . . **6-17**

Run Local Verification at Command Line . . . **6-19**
Specify Sources and Analysis Options Directly . . . **6-19**
Specify Sources and Analysis Options in Text File . . . **6-20**

Run Remote Analysis at the Command Line **6-21**
Run Remote Analysis **6-21**
Manage Remote Analysis **6-22**

Modularize Application at Command Line **6-26**
Basic Options **6-26**
Constrain Module Complexity During Partitioning **6-27**
Result Folder Names **6-27**
Forbid Cycles in Module Dependence Graph **6-28**

Scripts for Command-Line Verification **6-29**
Simple C Example **6-29**
Apache Example **6-29**
cxref Example **6-30**
T31 Example **6-30**
Dishwasher1 Example **6-30**
Satellite Example **6-31**

Create Command-Line Script from Project File **6-32**
Generate Scripting Files **6-32**
Run an Analysis **6-33**

Create Project Automatically from MATLAB Command Line **6-35**

Run Polyspace Analysis by Using MATLAB Scripts **6-37**
Specify Multiple Source Files **6-37**
Check for MISRA C:2012 Violations **6-38**
Check for Specific Defects or Coding Rule Violations **6-39**
Find Files That Do Not Compile **6-39**
Run Analysis on Cluster **6-40**

Generate MATLAB Scripts from Polyspace User Interface **6-41**

Troubleshoot Polyspace Analysis from MATLAB **6-44**

Storage of Temporary Files **6-46**

7 Troubleshooting Verification Problems

View Error Information When Analysis Stops **7-3**
View Error Information in User Interface **7-3**
View Error Information in Log File **7-4**

Troubleshoot Compilation and Linking Errors **7-7**
Issue **7-7**
Possible Cause: Deviations from ANSI C99 Standard **7-8**
Possible Cause: Linking Errors **7-9**
Possible Cause: Conflicts with Polyspace Function Stubs . . . **7-10**

Reduce Verification Time **7-12**
Issue **7-12**
Possible Cause: Temporary Folder on Network Drive **7-12**
Possible Cause: Large and Complex Application **7-12**
Possible Cause: Too Many Entry Points for Multitasking Applications **7-15**

Understand Verification Results **7-17**
Issue **7-17**
Possible Cause: Relation to Prior Code Operations **7-17**
Possible Cause: Software Assumptions **7-18**

Contact Technical Support **7-21**
Provide System Information **7-21**
Provide Information About the Issue **7-21**

Polyspace Cannot Find the Server **7-23**
Message **7-23**
Possible Cause **7-23**
Solution **7-23**

Job Manager Cannot Write to Database **7-24**
Message **7-24**
Possible Cause **7-24**
Workaround **7-24**

Undefined Identifier Error **7-26**
Issue **7-26**
Possible Cause: Missing Files **7-26**

Possible Cause: Unrecognized Keyword 7-26
Possible Cause: Declaration Embedded in #ifdef Statements 7-27
Possible Cause: Project Created from Non-Debug Build 7-28

Unknown Function Prototype Error 7-30
Issue 7-30
Cause 7-30
Solution 7-30

Error Related to #error Directive 7-32
Issue 7-32
Cause 7-32
Solution 7-32

Large Object Error 7-34
Issue 7-34
Cause 7-34
Solution 7-34

Errors Related to Generic Compiler 7-37
Issue 7-37
Cause 7-37
Solution 7-37

Errors Related to Keil or IAR Compiler 7-39
Missing Identifiers 7-39

Errors Related to Diab Compiler 7-40
Issue 7-40
Cause 7-40
Solution 7-40

Errors Related to TASKING Compiler 7-43
Issue 7-43
Cause 7-43
Solution 7-44

Errors from In-Class Initialization (C++) 7-45

Errors from Double Declarations of Standard Template Library Functions (C++) 7-46

Errors Related to GNU Compiler **7-47**
Issue **7-47**
Cause **7-47**
Solution **7-47**

Errors Related to Visual Compilers **7-48**
Import Folder **7-48**
pragma Pack **7-48**

Conflicting Declarations in Different Translation Units . . . **7-50**
Issue **7-50**
Possible Cause: Variable Declaration and Definition Mismatch **7-51**
Possible Cause: Function Declaration and Definition Mismatch **7-52**
Possible Cause: Macro-dependent Definitions **7-53**
Possible Cause: Keyword Redefined as Macro **7-54**
Possible Cause: Differences in Structure Packing **7-55**

Errors from Conflicts with Polyspace Header Files **7-56**
Issue **7-56**
Cause **7-56**
Solution **7-56**

C++ Standard Template Library Stubbing Errors **7-58**
Issue **7-58**
Cause **7-58**
Solution **7-58**

Lib C Stubbing Errors **7-59**
Extern C Functions **7-59**
Functional Limitations on Some Stubbed Standard ANSI Functions **7-60**

Errors from Assertion or Memory Allocation Functions . . . **7-61**
Issue **7-61**
Cause **7-61**
Solution **7-61**

Eclipse Java Version Incompatible with Polyspace Plug-in **7-62**
Issue **7-62**
Cause **7-62**

Solution 7-62

Reasons for Unchecked Code 7-64
Issue 7-64
Possible Cause: Compilation Errors 7-65
Possible Cause: Early Red or Gray Check 7-65
Possible Cause: Incorrect Options 7-68

Source Files or Functions Not Displayed in Results List . . . 7-69
Issue 7-69
Possible Cause: Files Not Verified 7-69
Possible Cause: Filters Applied 7-71

Incorrect Behavior of Standard Library Math Functions . . 7-73
Issue 7-73
Cause 7-73
Solution 7-73

Insufficient Memory During Report Generation 7-75
Message 7-75
Possible Cause 7-75
Solution 7-75

Errors with Temporary Files 7-76
No Access Rights 7-76
No Space Left on Device 7-76
Cannot Open Temporary File 7-77

Error from Special Characters 7-78
Issue 7-78
Cause 7-78
Workaround 7-78

Multiple File Error in File by File Verification 7-79
Issue 7-79
Possible Cause 7-79
Solution 7-79

Error from Disk Defragmentation and Antivirus Software 7-80
Issue 7-80
Possible Cause 7-80
Solution 7-80

License Error –4,0 7-81
Issue 7-81
Cause 7-81
Solution 7-81

8 Reviewing Verification Results

Result and Source Code Colors 8-3
Result Colors 8-3
Source Code Colors 8-6
Global Variable Colors 8-8

Review Red Checks 8-10
Step 1: Interpret Check Information 8-10
Step 2: Determine Root Cause of Check 8-11
Step 3: Look for Common Causes of Check 8-13

Review Gray Checks 8-15

Review Orange Checks 8-17
Step 1: Interpret Check Information 8-17
Step 2: Determine Root Cause of Check 8-19
Step 3: Look for Common Causes of Check 8-22
Step 4: Trace Check to Polyspace Assumption 8-22

Review Code Metrics 8-24
Impose Limits on Metrics 8-24
Comment and Justify Limit Violations 8-27

Review Global Variable Usage 8-29

Add Review Comments to Results 8-32
Assign and Save Comments 8-32
Import Review Comments from Previous Verifications 8-33

Justify Results Through Code Annotations 8-36
Add Annotations from the User Interface 8-36
Type Annotations Directly in Your Code 8-38
Syntax Examples 8-41

Add Review Comments to Code 8-44
Enter Code Comments in Specific Syntax 8-44
Copy Comment Syntax from Polyspace User Interface 8-48

Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results 8-50
Mapping Multiple MISRA C: 2004 Annotations to the Same MISRA C: 2012 Result 8-51

Define Custom Annotation Format 8-53
Define Annotation Syntax Format 8-56
Map Your Annotation to the Polyspace Annotation Syntax . . 8-60

Annotation Description Full XML Template 8-62
Example 8-66

Acronyms for Checks and Code Metrics 8-69
Checks 8-69
Code Complexity Metrics 8-70

Verification Following Red and Orange Checks 8-72
Code Following Red Check 8-73
Green Check Following Orange Check 8-73
Gray Check Following Orange Check 8-74
Red Check Following Orange Check 8-75

Types of Run-Time Checks 8-77
Data Flow Checks 8-77
Numerical Checks 8-78
Static Memory Checks 8-78
Control Flow Checks 8-78
C++ Checks 8-79
Other Checks 8-80

HIS Code Complexity Metrics 8-81
Project 8-81
File 8-81
Function 8-81

Result Views in Polyspace User Interface 8-83
Dashboard 8-83
Results List 8-88
Source 8-91

Result Details **8-99**
Call Hierarchy **8-102**
Variable Access **8-104**

Filter and Group Results **8-113**
Filter Results **8-113**
Group Results **8-117**

Prioritize Check Review **8-119**

Software Quality Objectives **8-122**
Comparing Verification Results Against Software Quality Objectives **8-129**

Project and Results Folder Contents **8-131**
Files in the Results Folder **8-131**

Generate Report **8-133**
Specify Report Generation Before Verification **8-133**
Generate Report After Verification **8-134**

Export Polyspace Analysis Results **8-136**
Export Results to Text File **8-136**
Export Results to MATLAB Table **8-138**
View Exported Results **8-138**

Visualize Polyspace Analysis Results in MATLAB **8-140**
Export Results to MATLAB Table **8-140**
Generate Graphs from Results and Include in Report **8-140**

Customize Existing Report Template **8-144**
Prerequisites **8-144**
View Components of Template **8-144**
Change Components of Template **8-146**
Further Exploration **8-148**

Sample Report Template Customizations **8-150**
Add List of Recursive Functions **8-150**
Show Red Run-Time Checks Only **8-151**
Show Non-Justified Run-Time Checks Only **8-152**
Add Chapter for Functional Design Errors **8-152**

Set Character Encoding Preferences **8-154**

9 Reviewing Checks

Review and Fix Absolute Address Usage Checks **9-3**

Review and Fix Correctness Condition Checks **9-4**
Step 1: Interpret Check Information **9-4**
Step 2: Determine Root Cause of Check **9-7**
Step 3: Trace Check to Polyspace Assumption **9-9**

Review and Fix Division by Zero Checks **9-10**
Step 1: Interpret Check Information **9-10**
Step 2: Determine Root Cause of Check **9-11**
Step 3: Look for Common Causes of Check **9-14**
Step 4: Trace Check to Polyspace Assumption **9-14**

Review and Fix Function Not Called Checks **9-16**
Step 1: Interpret Check Information **9-16**
Step 2: Determine Root Cause of Check **9-16**
Step 3: Look for Common Causes of Check **9-17**

Review and Fix Function Not Reachable Checks **9-19**
Step 1: Interpret Check Information **9-19**
Step 2: Determine Root Cause of Check **9-19**

Review and Fix Function Not Returning Value Checks **9-21**
Step 1: Interpret Check Information **9-21**
Step 2: Determine Root Cause of Check **9-21**

Review and Fix Illegally Dereferenced Pointer Checks **9-23**
Step 1: Interpret Check Information **9-23**
Step 2: Determine Root Cause of Check **9-26**
Step 3: Look for Common Causes of Check **9-28**
Step 4: Trace Check to Polyspace Assumption **9-29**

Review and Fix Incorrect Object Oriented Programming Checks **9-31**
Step 1: Interpret Check Information **9-31**
Step 2: Determine Root Cause of Check **9-32**

Review and Fix Invalid C++ Specific Operations Checks .. **9-34**
Step 1: Interpret Check Information **9-34**

Step 2: Determine Root Cause of Check **9-35**
Step 3: Trace Check to Polyspace Assumption **9-36**

Review and Fix Invalid Shift Operations Checks **9-37**
Step 1: Interpret Check Information **9-37**
Step 2: Determine Root Cause of Check **9-38**
Step 3: Look for Common Causes of Check **9-41**
Step 4: Trace Check to Polyspace Assumption **9-41**

Review and Fix Invalid Use of Standard Library Routine Checks **9-43**
Step 1: Interpret Check Information **9-43**
Step 2: Trace Check to Polyspace Assumption **9-45**

Invalid Use of Standard Library Floating Point Routines . . **9-46**
What the Check Looks For **9-46**
Single-Argument Functions Checked **9-47**
Functions with Multiple Arguments **9-48**

Review and Fix Non-initialized Local Variable Checks **9-50**
Step 1: Interpret Check Information **9-50**
Step 2: Determine Root Cause of Check **9-51**
Step 3: Look for Common Causes of Check **9-52**
Step 4: Trace Check to Polyspace Assumption **9-53**

Review and Fix Non-initialized Pointer Checks **9-54**
Step 1: Interpret Check Information **9-54**
Step 2: Determine Root Cause of Check **9-54**
Step 3: Trace Check to Polyspace Assumption **9-56**

Review and Fix Non-initialized Variable Checks **9-57**
Step 1: Interpret Check Information **9-57**
Step 2: Determine Root Cause of Check **9-58**
Step 3: Trace Check to Polyspace Assumption **9-58**

Review and Fix Non-Terminating Call Checks **9-60**
Step 1: Determine Root Cause of Check **9-60**
Step 2: Look for Common Causes of Check **9-61**

Identify Function Call with Run-Time Error **9-63**

Review and Fix Non-Terminating Loop Checks **9-65**
Step 1: Interpret Check Information **9-65**

Step 2: Determine Root Cause of Check **9-65**
Step 3: Look for Common Causes of Check **9-67**

Identify Loop Operation with Run-Time Error **9-69**

Review and Fix Null This-pointer Calling Method Checks . **9-72**
Step 1: Interpret Check Information **9-72**
Step 2: Determine Root Cause of Check **9-73**

Review and Fix Out of Bounds Array Index Checks **9-74**
Step 1: Interpret Check Information **9-74**
Step 2: Determine Root Cause of Check **9-75**
Step 3: Look for Common Causes of Check **9-77**
Step 4: Trace Check to Polyspace Assumption **9-77**

Review and Fix Overflow Checks **9-79**
Step 1: Interpret Check Information **9-79**
Step 2: Determine Root Cause of Check **9-80**
Step 3: Look for Common Causes of Check **9-83**
Step 4: Trace Check to Polyspace Assumption **9-83**

Detect Overflows in Buffer Size Computation **9-84**

Review and Fix Return Value Not Initialized Checks **9-86**
Step 1: Interpret Check Information **9-86**
Step 2: Determine Root Cause of Check **9-86**
Step 3: Look for Common Causes of Check **9-88**
Step 4: Trace Check to Polyspace Assumption **9-88**

Review and Fix Uncaught Exception Checks **9-90**
Step 1: Interpret Check Information **9-90**
Step 2: Determine Root Cause of Check **9-90**

Review and Fix Unreachable Code Checks **9-93**
Step 1: Interpret Check Information **9-93**
Step 2: Determine Root Cause of Check **9-94**
Step 3: Look for Common Causes of Check **9-96**

Review and Fix User Assertion Checks **9-99**
Step 1: Determine Root Cause of Check **9-99**
Step 2: Look for Common Causes of Check **9-102**
Step 3: Trace Check to Polyspace Assumption **9-102**

Find Relations Between Variables in Code **9-104**
Insert Pragma to Determine Variable Relation **9-104**
Further Exploration **9-106**

10 Managing Orange Checks

Sources of Orange Checks **10-2**
When Orange Checks Occur **10-2**
Why Review Orange Checks **10-2**
How to Review Orange Checks **10-3**
How to Reduce Orange Checks **10-3**

Managing Orange Checks **10-5**
Software Development Stage **10-6**
Quality Goals **10-9**

Limit Display of Orange Checks **10-10**

Critical Orange Checks **10-13**
Path **10-13**
Bounded Input Values **10-14**
Unbounded Input Values **10-15**

Reduce Orange Checks **10-16**
Provide Context for Verification **10-16**
Improve Verification Precision **10-17**
Follow Coding Rules **10-17**

Test Orange Checks for Run-Time Errors **10-19**
Run Tests for Full Range of Input **10-19**
Run Tests for Specified Range of Input **10-22**

Limitations of Automatic Orange Tester **10-24**
Unsupported Platforms **10-24**
Unsupported Polyspace Options **10-24**
Options with Restrictions **10-24**
Unsupported C Routines **10-25**

11 Coding Rule Sets and Concepts

Rule Checking **11-2**
Polyspace Coding Rule Checker **11-2**
Differences Between Bug Finder and Code Prover **11-2**

Polyspace MISRA C 2004 and MISRA AC AGC Checkers . . . **11-4**

Software Quality Objective Subsets (C:2004) **11-5**
Rules in SQO-Subset1 . . . . **11-5**
Rules in SQO-Subset2 . . . . **11-6**

Software Quality Objective Subsets (AC AGC) **11-10**
Rules in SQO-Subset1 . . . . **11-10**
Rules in SQO-Subset2 . . . . **11-11**

MISRA C:2004 and MISRA AC AGC Coding Rules **11-14**
Supported MISRA C:2004 and MISRA AC AGC Rules **11-14**
Unsupported MISRA C:2004 and MISRA AC AGC Rules . . **11-53**

Polyspace MISRA C:2012 Checker **11-56**

Software Quality Objective Subsets (C:2012) **11-58**
Guidelines in SQO-Subset1 . . . . **11-58**
Guidelines in SQO-Subset2 . . . . **11-59**

Coding Rule Subsets Checked Early in Analysis **11-63**
MISRA C: 2004 and MISRA AC AGC Rules **11-63**
MISRA C: 2012 Rules **11-73**

Unsupported MISRA C:2012 Guidelines **11-83**

Polyspace MISRA C++ Checker **11-84**

Software Quality Objective Subsets (C++) **11-85**
SQO Subset 1 – Direct Impact on Selectivity **11-85**
SQO Subset 2 – Indirect Impact on Selectivity **11-87**

MISRA C++ Coding Rules **11-92**
Supported MISRA C++ Coding Rules **11-92**
Unsupported MISRA C++ Rules **11-115**

Polyspace JSF C++ Checker **11-120**

JSF C++ Coding Rules **11-121**
Supported JSF C++ Coding Rules **11-121**
Unsupported JSF++ Rules **11-145**

12 Checking Coding Rules

Set Up Coding Rules Checking **12-2**
Select Predefined Coding Rule Sets **12-2**
Select Specific MISRA or JSF Coding Rules **12-3**
Create Coding Rules **12-4**

Create Custom Coding Rules **12-6**

Format of Custom Coding Rules File **12-8**

Exclude Files from Rules Checking **12-10**

Configure Additional Options for Certain Rules **12-11**
Specify Allowed Custom Pragma Directives **12-11**
Specify Effective Boolean Types **12-11**

Review Coding Rule Violations **12-13**

Filter and Group Coding Rule Violations **12-15**
Filter Coding Rules **12-15**
Group Coding Rules **12-15**
Suppress Certain Rules from Display in One Click **12-15**

Generate Coding Rules Report **12-18**

Coding Rules Not Checked in Compilation Phase **12-20**

13 Software Quality with Polyspace Metrics

Code Quality Metrics **13-2**
Summary Tab **13-2**
Code Metrics Tab **13-5**
Coding Rules Tab **13-6**
Run-Time Checks Tab **13-7**

Generate Code Quality Metrics **13-12**
Upload Results to Polyspace Metrics **13-12**
Specify Automatic Uploading of Results **13-13**

View Code Quality Metrics **13-19**
Open Metrics Interface **13-19**
View All Projects and Runs **13-19**
Review Metrics for Particular Project or Run **13-21**

Compare Metrics Against Software Quality Objectives . . . **13-23**
Apply Predefined Objectives to Metrics **13-23**
Customize Software Quality Objectives **13-25**

View Trends in Code Quality Metrics **13-29**

Web Browser Requirements for Polyspace Metrics **13-32**

Elements in Custom Software Quality Objectives File **13-33**
HIS Metrics **13-33**
Non-HIS Metrics **13-34**

14 Configure Model for Code Analysis

Configure Simulink Model **14-2**

Recommended Model Settings for Code Analysis **14-3**

Check Simulink Model Settings **14-6**
Check Simulink Model Settings Using the Code Generation Advisor **14-6**
Check Simulink Model Settings Before Analysis **14-7**
Check Simulink Model Settings Automatically **14-9**

Annotate Blocks for Known Results **14-12**

15 Polyspace Code Prover Analysis in Simulink

Install Polyspace Plugin for Simulink **15-2**

Verify Generated Code Using Polyspace Code Prover **15-4**

Verify Code Generated from Simulink Subsystem **15-7**
Open Model **15-7**
Generate Code **15-8**
Verify Code **15-9**
Review Verification Results **15-9**
Trace Errors Back to Model and Fix Them **15-10**
Check for Coding Rule Violations **15-13**
Annotate Blocks to Justify Results **15-13**

Auto-Annotate Generated Code to Justify Checks **15-16**

Main Generation for Model Verification **15-18**

Configure Data Range Settings **15-20**

Embedded Coder Considerations **15-22**
Default Options **15-22**
Data Range Specification **15-22**
Recommended Polyspace options for Verifying Generated Code **15-23**
Hardware Mapping Between Simulink and Polyspace **15-26**

TargetLink Considerations **15-27**
TargetLink Support **15-27**
Default Options **15-27**

Lookup Tables 15-28
Data Range Specification 15-28
Code Generation Options 15-29
Justifying Results in Model 15-29

View Results in Polyspace Code Prover 15-30

Identify Errors in Simulink Models 15-31

Troubleshoot Back to Model 15-34
Back-to-Model Links Do Not Work 15-34
Your Model Already Uses Highlighting 15-34

16 Configure Code Analysis Options

Polyspace Configuration for Generated Code 16-2

Include Handwritten Code 16-3

Configure Analysis Depth for Referenced Models 16-4

Configure Advanced Polyspace Analysis Options 16-5
Set Advanced Analysis Options 16-5
Use a Saved Polyspace Configuration File Template 16-6
Reset Polyspace Options for a Simulink Model 16-7

Set Custom Target Settings 16-8

Set Up Remote Analysis 16-11

Manage Results 16-12
Open Polyspace Results Automatically 16-12
Specify Location of Results 16-13
Save Results to a Simulink Project 16-14

Specify Signal Ranges 16-15
Specify Signal Range Through Source Block Parameters 16-15
Specify Signal Range Through Base Workspace 16-17

17 Run Polyspace on Generated Code

Specify Type of Analysis to Perform **17-2**

Run Analysis for Embedded Coder **17-3**
Start the Analysis **17-3**
Monitor Progress **17-4**

Run Analysis for TargetLink **17-5**
Start the Analysis **17-5**
Monitor Progress **17-6**

Verify S-Function Code **17-7**
S-Function Analysis Workflow **17-7**
Compile S-Functions to Be Compatible with Polyspace **17-7**
Example S-Function Analysis **17-8**

18 Using Polyspace Software in the Eclipse IDE

Install Polyspace Plugin for Eclipse **18-2**
Install Polyspace Plugin for Eclipse IDE **18-2**
Uninstall Polyspace Plugin for Eclipse IDE **18-4**

Configure Verification **18-5**
Prerequisites **18-5**
Eclipse Directly Refers to Your Compilation Toolchain **18-5**
Eclipse Uses Your Compilation Toolchain Through Build Command **18-7**
Next Steps **18-8**

Run Verification **18-9**
Prerequisites **18-9**
Start, Monitor and Stop Verification **18-9**
Next Steps **18-10**

Review Results **18-11**
Prerequisites **18-11**

Review Results **18-11**
Save Multiple Results **18-11**
Limit Display of Results **18-12**

Glossary

Introduction to Polyspace Products

- "Polyspace Verification" on page 1-2
- "How Polyspace Verification Works" on page 1-5
- "Related Products" on page 1-7

Polyspace Verification

In this section...
"Polyspace Verification" on page 1-2
"Value of Polyspace Verification" on page 1-2

Polyspace Verification

Polyspace products verify C, C++, and Ada code by detecting run-time errors before code is compiled and executed.

To verify the source code, you set up verification parameters in a project, run the verification, and review the results. A graphical user interface helps you to efficiently review verification results. The software assigns a color to operations in the source code as follows:

- **Green** – Indicates that the operation is proven to not have certain kinds of error.
- **Red** – Indicates that the operation is proven to have at least one error.
- **Gray** – Indicates unreachable code.
- **Orange** – Indicates that the operation can have an error along some execution paths.

The color-coding helps you to quickly identify errors and find the exact location of an error in the source code. After you fix errors, you can easily run the verification again.

Value of Polyspace Verification

Polyspace verification can help you to:

- "Enhance Software Reliability" on page 1-2
- "Decrease Development Time" on page 1-3
- "Improve the Development Process" on page 1-4

Enhance Software Reliability

Polyspace software enhances the reliability of your C/C++ applications by proving code correctness and identifying run-time errors. Using advanced verification techniques, Polyspace software performs an exhaustive verification of your source code.

Because Polyspace software verifies all executions of your code, it can identify code that:

- Never has an error
- Always has an error
- Is unreachable
- Might have an error

With this information, you know how much of your code does not contain run-time errors, and you can improve the reliability of your code by fixing errors.

You can also improve the quality of your code by using Polyspace verification software to check that your code complies with established coding standards, such as the MISRA C®, MISRA® C++ or JSF® C++ standards.[1]

Decrease Development Time

Polyspace software reduces development time by automating the verification process and helping you to efficiently review verification results. You can use it at any point in the development process. However, using it during early coding phases allows you to find errors when it is less costly to fix them.

You use Polyspace software to verify source code before compile time. To verify the source code, you set up verification parameters in a project, run the verification, and review the results. This process takes significantly less time than using manual methods or using tools that require you to modify code or run test cases.

Color-coding of results helps you to quickly identify errors. You will spend less time debugging because you can see the exact location of an error in the source code. After you fix errors, you can easily run the verification again.

Polyspace verification software helps you to use your time effectively. Because you know the parts of your code that do not have errors, you can focus on the code with proven (red code) or potential errors (orange code).

Reviewing code that might have errors (orange code) can be time-consuming, but Polyspace software helps you with the review process. You can use filters to focus on certain types of errors or you can allow the software to identify the code that you should review.

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

Improve the Development Process

Polyspace software makes it easy to share verification parameters and results, allowing the development team to work together to improve product reliability. Once verification parameters have been set up, developers can reuse them for other files in the same application.

Polyspace verification software supports code verification throughout the development process:

- An individual developer can find and fix run-time errors during the initial coding phase.
- Quality assurance engineers can check overall reliability of an application.
- Managers can monitor application reliability by generating reports from the verification results.

How Polyspace Verification Works

Polyspace software uses *static verification* to prove the absence of run-time errors. Static verification derives the dynamic properties of a program without actually executing it. This differs significantly from other techniques, such as run-time debugging, in that the verification it provides is not based on a given test case or set of test cases. The dynamic properties obtained in the Polyspace verification are true for all executions of the software.

What is Static Verification

Static verification is a broad term, and is applicable to any tool that derives dynamic properties of a program without executing the program. However, most static verification tools only verify the complexity of the software, in a search for constructs that may be potentially erroneous. Polyspace verification provides deep-level verification identifying almost all run-time errors and possible access conflicts with global shared data.

Polyspace verification works by approximating the software under verification, using representative approximations of software operations and data.

For example, consider the following code:

```
for (i=0 ; i<1000 ; ++i)
{    tab[i] = foo(i);
}
```

To check that the variable `i` never overflows the range of `tab`, a traditional approach would be to enumerate each possible value of `i`. One thousand checks would be required.

Using the static verification approach, the variable `i` is modelled by its domain variation. For instance, the model of `i` is that it belongs to the static interval [0..999]. (Depending on the complexity of the data, convex polyhedrons, integer lattices and more elaborate models are also used for this purpose).

By definition, an approximation leads to information loss. For instance, the information that `i` is incremented by one every cycle in the loop is lost. However, the important fact is that this information is not required to ensure that no range error will occur; it is only necessary to prove that the domain variation of `i` is smaller than the range of `tab`. Only one check is required to establish that — and hence the gain in efficiency compared to traditional approaches.

Static code verification has an exact solution. However, this exact solution is not practical, as it would require the enumeration of all possible test cases. As a result, approximation is required for a usable tool.

Exhaustiveness

Nothing is lost in terms of exhaustiveness. The reason is that Polyspace verification works by performing upper approximations. In other words, the computed variation domain of a program variable is a superset of its actual variation domain. As a result, Polyspace verifies run-time error items that require checking.

Related Products

In this section...
“Polyspace Bug Finder” on page 1-7
“Polyspace Products for Verifying Ada Code” on page 1-7
“Tool Qualification and Certification” on page 1-7

Polyspace Bug Finder

For information about Polyspace Bug Finder™ , see `http://www.mathworks.com/products/polyspace-bug-finder/`.

Polyspace Products for Verifying Ada Code

For information about Polyspace products that verify Ada code, see the following:

`http://www.mathworks.com/products/polyspaceclientada/`

`http://www.mathworks.com/products/polyspaceserverada/`

Tool Qualification and Certification

You can use the DO Qualification Kit and IEC Certification Kit products to qualify Polyspace Products for C/C++ for DO and IEC Certification.

To view the artifacts available with these kits, use the Certification Artifacts Explorer. Artifacts included in the kits are not accessible from the MathWorks® web site.

For more information on the IEC Certification Kit, see IEC Certification Kit (for ISO 26262 and IEC 61508).

For more information on the DO Qualification Kit, see DO Qualification Kit (for DO-178).

2

How to Use Polyspace Software

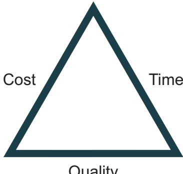
- "Polyspace Verification and the Software Development Cycle" on page 2-2
- "Implement Process for Verification" on page 2-4
- "Sample Workflows for Polyspace Verification" on page 2-6
- "Define Your Requirements" on page 2-19

Polyspace Verification and the Software Development Cycle

In this section...
“Software Quality and Productivity” on page 2-2
“Best Practices for Verification Workflow” on page 2-3

Software Quality and Productivity

The goal of most software development teams is to maximize both quality and productivity. However, when developing software, there are three related variables to consider: cost, quality, and time.



Changing the requirements for one of these variables affects the other two.

Generally, the criticality of your application determines the balance between these three variables – your quality model. With classical testing processes, development teams generally try to achieve their quality model by testing all modules in an application until each module meets the required quality level. Unfortunately, this process often ends before quality requirements are met, because the available time or budget has been exhausted.

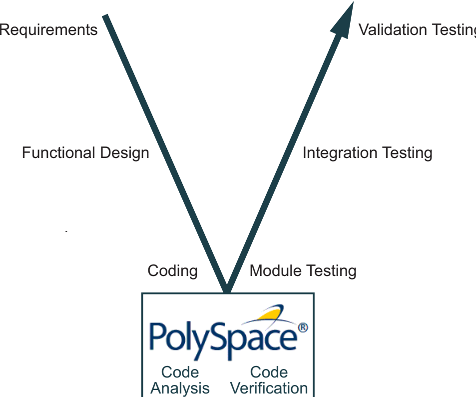
Polyspace verification allows a different process. Polyspace verification can support both productivity improvement and quality improvement at the same time. However, you must balance the aims of these activities.

You should not perform code verification at the end of the development process. To achieve maximum quality and productivity, integrate verification into your development process, considering time and cost restrictions.

This section describes how to integrate Polyspace verification into your software development cycle. It explains both how to use Polyspace verification in your current development process, and how to change your process to get more out of verification.

Best Practices for Verification Workflow

Polyspace verification can be used throughout the software development cycle. However, to maximize both quality and productivity, the most efficient time to use it is early in the development cycle.



Polyspace Verification in the Development Cycle

Typically, verification is conducted in two stages. First, you verify code as it is written, to check coding rules and quickly identify obvious defects. Once the code is stable, you verify it again before module/unit testing, with more stringent verification and review criteria.

Using verification early in the development cycle improves both quality and productivity, because it allows you to find and manage defects soon after the code is written. This saves time because each user is familiar with their own code, and can quickly determine why the code contains defects. In addition, defects are cheaper to fix at this stage, since they can be addressed before the code is integrated into a larger system.

Implement Process for Verification

In this section...
"Overview of the Polyspace Process" on page 2-4
"Define Process to Meet Your Goals" on page 2-4
"Apply Process to Assess Code Quality" on page 2-5
"Improve Your Verification Process" on page 2-5

Overview of the Polyspace Process

Polyspace verification cannot magically produce quality code at the end of the development process. However, if you integrate Polyspace verification into your development process, Polyspace verification helps you to measure the quality of your code, identify issues, and ultimately achieve your own quality goals.

To implement Polyspace verification within your development process, you must perform each of the following steps:

1. Define your quality goals.
2. Define a process to match your quality goals.
3. Apply the process to assess the quality of your code.
4. Improve the process.

Define Process to Meet Your Goals

Once you have defined your quality goals, you must define a process that allows you to meet those goals. Defining the process involves actions both within and outside Polyspace software.

These actions include:

- Communicating coding standards (coding rules) to your development team.
- Setting Polyspace analysis options. For more information, see "Specify Analysis Options" on page 3-22.
- Setting review criteria for consistent review of results. For more information, see "Limit Display of Orange Checks" on page 10-10.

Apply Process to Assess Code Quality

Once you have defined a process that meets your quality goals, it is up to your development and testing teams to apply it consistently to all software components.

This process includes:

1. Running a Polyspace verification on each software component as it is written.
2. Reviewing verification results consistently. See "Add Review Comments to Results" on page 8-32.
3. Saving review comments for each component, so they are available for future review. See "Import Review Comments from Previous Verifications" on page 8-33.
4. Performing additional verifications on each component, as defined by your quality goals.

Improve Your Verification Process

Once you review initial verification results, you can assess both the overall quality of your code, and how well the process meets your requirements for software quality, development time, and cost restrictions.

Based on these factors, you may want to take actions to modify your process. These actions may include:

- Reassessing your quality goals.
- Changing your development process to produce code that is easier to verify.
- Changing Polyspace analysis options to improve the precision of the verification.
- Changing Polyspace options to change how verification results are reported.

For more information, see "Reduce Orange Checks" on page 10-16.

Sample Workflows for Polyspace Verification

In this section...

"Overview of Verification Workflows" on page 2-6
"Software Developers and Testers – Standard Development Process" on page 2-6
"Software Developers and Testers – Rigorous Development Process" on page 2-9
"Quality Engineers – Code Acceptance Criteria" on page 2-12
"Quality Engineers – Certification/Qualification" on page 2-14
"Model-Based Design Users — Verifying Generated Code" on page 2-15
"Project Managers — Integrating Polyspace Verification with Configuration Management Tools" on page 2-18

Overview of Verification Workflows

Polyspace verification supports two goals at the same time:

- Reducing the cost of testing and validation
- Improving software quality

You can use Polyspace verification in different ways depending on your development context and quality model.

This section provides sample workflows that show how to use Polyspace verification in a variety of development contexts.

Software Developers and Testers – Standard Development Process

User Description

This workflow applies to software developers and test groups using a standard development process, where coding rules are not used or followed consistently.

Quality

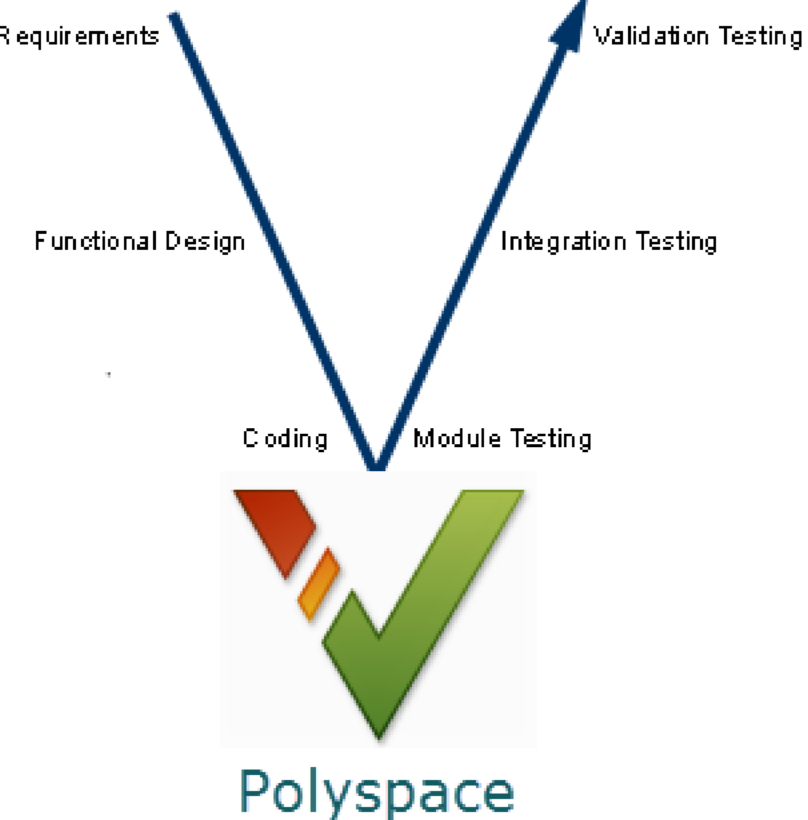
The main goal of Polyspace verification is to improve productivity while maintaining or improving software quality. Verification helps developers and testers find and fix bugs

more quickly than other processes. It also improves software quality by identifying bugs that otherwise might remain in the software.

In this process, the goal is not to completely prove the absence of errors. The goal is to deliver code of equal or better quality that other processes, while optimizing productivity to provide a predictable time frame with minimal delays and costs.

Verification Workflow

This process involves file-by-file verification immediately after coding, and again just before functional testing.



The verification workflow consists of the following steps:

1. The project leader configures a Polyspace project to perform robustness verification, using default Polyspace options.

   Note This means that verification uses the automatically generated "main" function. This main will call unused procedures and functions with full range parameters.

   ---
2. Each developer performs file-by-file verification as they write code, and reviews verification results.
3. The developer fixes **red** errors and examines **gray** code identified by the verification.
4. Until coding is complete, the developer repeats steps 2 and 3 as required..
5. Once a developer considers a file complete, they perform a final verification.
6. The developer fixes **red** errors, examines **gray** code, and performs a selective orange review.

   Note The goal of the selective orange review is to find as many bugs as possible within a limited period of time.

Using this approach, it is possible that some bugs may remain in unchecked oranges. However, the verification process represents a significant improvement from other testing methods.

Costs and Benefits

When using verification to detect bugs:

- **Red and gray checks** – Reviewing red and gray checks provides a quick method to identify real run-time errors in the code.
- **Orange checks** – Selective orange review provides a method to identify potential run-time errors as quickly as possible. The time required to find one bug varies from 5 minutes to 1 hour, and is typically around 30 minutes. This represents an average of two minutes per orange check review, and a total of 20 orange checks per package in Ada and 60 orange checks per file in C.

Disadvantages to this approach:

- **Number of orange checks –** If you do not use coding rules, your verification results will contain more orange checks.
- **Unreviewed orange checks** – Some bugs may remain in unchecked oranges.

Software Developers and Testers – Rigorous Development Process

User Description

This workflow applies to software developers and test engineers working within development groups. These users are often developing software for embedded systems, and typically use coding rules.

These users typically want to find bugs early in the development cycle using a tool that is fast and iterative.

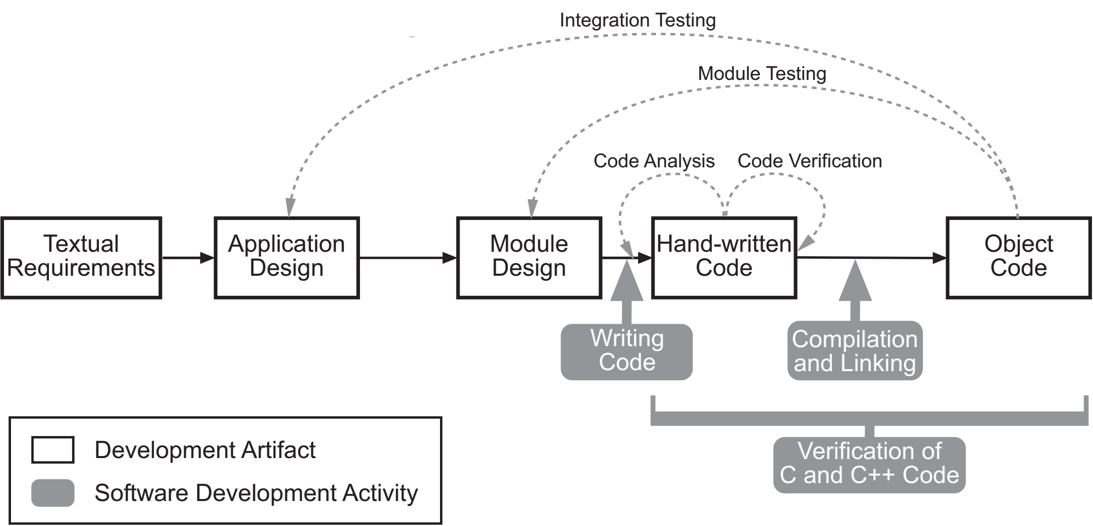
Quality

The goal of Polyspace verification is to improve software quality with equal or increased productivity.

Verification can prove the absence of run-time errors, while helping developers and testers to find and fix defects efficiently.

Verification Workflow

This process involves both code analysis and code verification during the coding phase, and thorough review of verification results before module testing. It may also involve integration analysis before integration testing.



Workflow for Code Verification

Note Solid arrows in the figure indicate the progression of software development activities.

The verification workflow consists of the following steps:

1. The project leader configures a Polyspace project to perform contextual verification. This involves:

   - Using Data Range Specifications (DRS) to define initialization ranges for input data. For example, if a variable “x” is read by functions in the file, and if x can be initialized to any value between 1 and 10, this information should be included in the DRS file.
   - Creates a “main” program to model call sequence, instead of using the automatically generated main.
   - Sets options to check the properties of some output variables. For example, if a variable “y” is returned by a function in the file and should always be returned with a value in the range 1 to 100, then Polyspace can flag instances where that range of values might be breached.

2 The project leader configures the project to check the required coding rules.
3 Each developer performs file-by-file verification as they write code, and reviews both coding rule violations and verification results.
4 The developer fixes coding rule violations and **red** errors, examines **gray** code, and performs a selective orange review.
5 Until coding is complete, the developer repeats steps 2 and 3 as required.
6 Once a developer considers a file complete, they perform a final verification.
7 The developer or tester performs an exhaustive orange review on the remaining orange checks.

Note The goal of the exhaustive orange review is to examine orange checks that are not reviewed as part of selective reviews. When you fix coding rule violations, the total number of orange checks is reduced, and the remaining orange checks are likely to reveal problems with the code.

Optionally, an additional verification can be performed during the integration phase. The purpose of this additional verification is to track integration bugs, and review:

- Red and gray integration checks;
- The remaining orange checks with a selective review: *Integration bug tracking*.

Costs and Benefits

With this approach, Polyspace verification typically provides the following benefits:

- Fewer orange checks in the verification results (improved selectivity). The number of orange checks is typically reduced to 3–5 per file, yielding an average of 1 bug. Often, several of the orange checks represent the same bug.
- Fewer gray checks in the verification results.
- Typically, each file requires two verifications before it can be checked-in to the configuration management system.
- The average verification time is about 15 minutes.

Note If the development process includes data rules that determine the data flow design, the benefits might be greater. Using data rules reduces the potential of verification finding integration bugs.

If performing the optional verification to find integration bugs, you may see the following results. On a typical 50,000 line project:

- A selective orange review may reveal **one integration bug per hour** of code review.
- Selective orange review takes about 6 hours to complete. This is long enough to review orange checks throughout the whole application and represents a step towards an exhaustive orange check review. Spending more time is unlikely to be efficient.
- An exhaustive orange review would take between 4 and 6 days, assuming that 50,000 lines of code contains approximately 400–800 orange checks. Exhaustive orange review is typically recommended only for high-integrity code, where the consequences of a potential error justify the cost of the review.

Quality Engineers – Code Acceptance Criteria

User Description

This workflow applies to quality engineers who work outside of software development groups, and are responsible for independent verification of software quality and adherence to standards.

These users generally receive code late in the development cycle, and may even be verifying code that is written by outside suppliers or other external companies. They are concerned with not just detecting bugs, but measuring quality over time, and developing processes to measure, control, and improve product quality going forward.

Quality

The main goal of Polyspace verification is to control and evaluate the safety of an application.

The criteria used to evaluate code can vary widely depending on the nature of the application. For example:

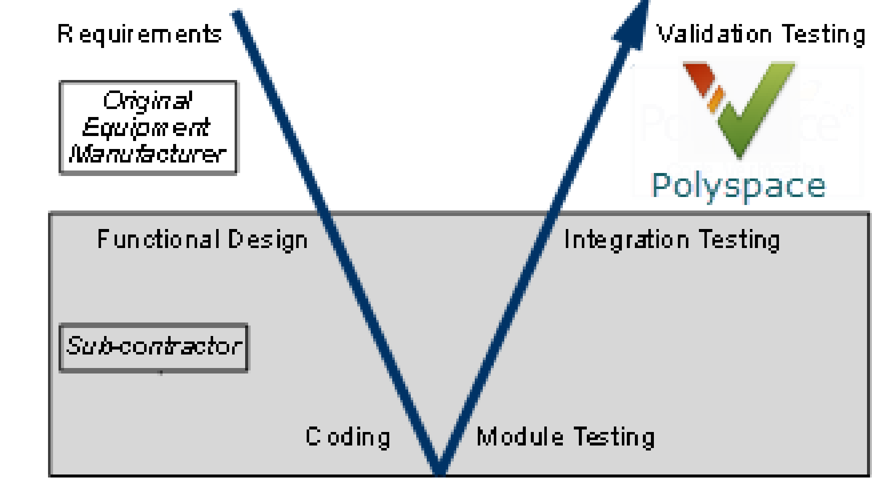
- You may be satisfied with zero red checks.
- In addition to zero red checks, you may want to conduct an exhaustive orange check review.

Typically, these criteria become increasingly stringent as a project advances from early, to intermediate, and eventually to final delivery.

For more information on defining these criteria, see “Customize Software Quality Objectives” on page 13-25.

Verification Workflow

This process usually involves both code analysis and code verification before validation phase, and thorough review of verification results based on defined quality goals.



Note Verification is often performed multiple times, as multiple versions of the software are delivered.

The verification workflow consists of the following steps:

1. Quality engineering group defines clear quality goals for the code to be written, including specific quality levels for each version of the code to be delivered (first, intermediate, or final delivery) For more information, see “Customize Software Quality Objectives” on page 13-25.
2. Development group writes code according to established standards.
3. Development group delivers software to the quality engineering group.
4. The project leader configures the Polyspace project to meet the defined quality goals, as described in “Define Process to Meet Your Goals” on page 2-4.
5. Quality engineers perform verification on the code.

6 Quality engineers review **red** errors, **gray** code, and the number of orange checks defined in the process.

Note The number of orange checks reviewed often depends on the version of software being tested (first, intermediate, or final delivery). This can be defined by quality level (see “Define Broad Requirements for Verification” on page 2-19).

---

7 Quality engineers create reports documenting the results of the verification, and communicate those results to the supplier.

8 Quality engineers repeat steps 5–7 for each version of the code delivered.

Costs and Benefits

The benefits of code verification at this stage are the same as with other verification processes, but the cost of correcting faults is higher, because verification takes place late in the development cycle.

It is possible to perform an exhaustive orange review at this stage, but the cost of doing so can be high. If you want to review all orange checks at this phase, it is important to use development and verification processes that minimize the number of orange checks. This includes:

- Developing code using strict coding and data rules.
- Providing accurate manual stubs for unresolved function calls.
- Using DRS to provide accurate data ranges for input variables.

Taking these steps will minimize the number of orange checks reported by the verification, and make it more likely that remaining orange checks represent real issues with the software.

Quality Engineers – Certification/Qualification

User Description

This workflow applies to quality engineers who work with applications requiring outside quality certification, such as IEC 61508 certification or DO-178 qualification.

These users must perform a set of activities to meet certification requirements.

You can use the “IEC Certification Kit (for ISO 26262 and IEC 61508)” to help qualify Polyspace products within an IEC 61508, ISO 26262, EN 50128, or other related functional-safety standard certification environment.

You can use the “DO Qualification Kit (for DO-178)” to help qualify Polyspace products within an DO-178 qualification environment.

Model-Based Design Users — Verifying Generated Code

User Description

This workflow applies to users who have adopted model-based design to generate code for embedded application software.

These users generally use Polyspace software in combination with several other MathWorks products, including Simulink®, Embedded Coder® , and Simulink Design Verifier™ products. In many cases, these customers combine application components that are manually written code with those created using generated code.

Quality

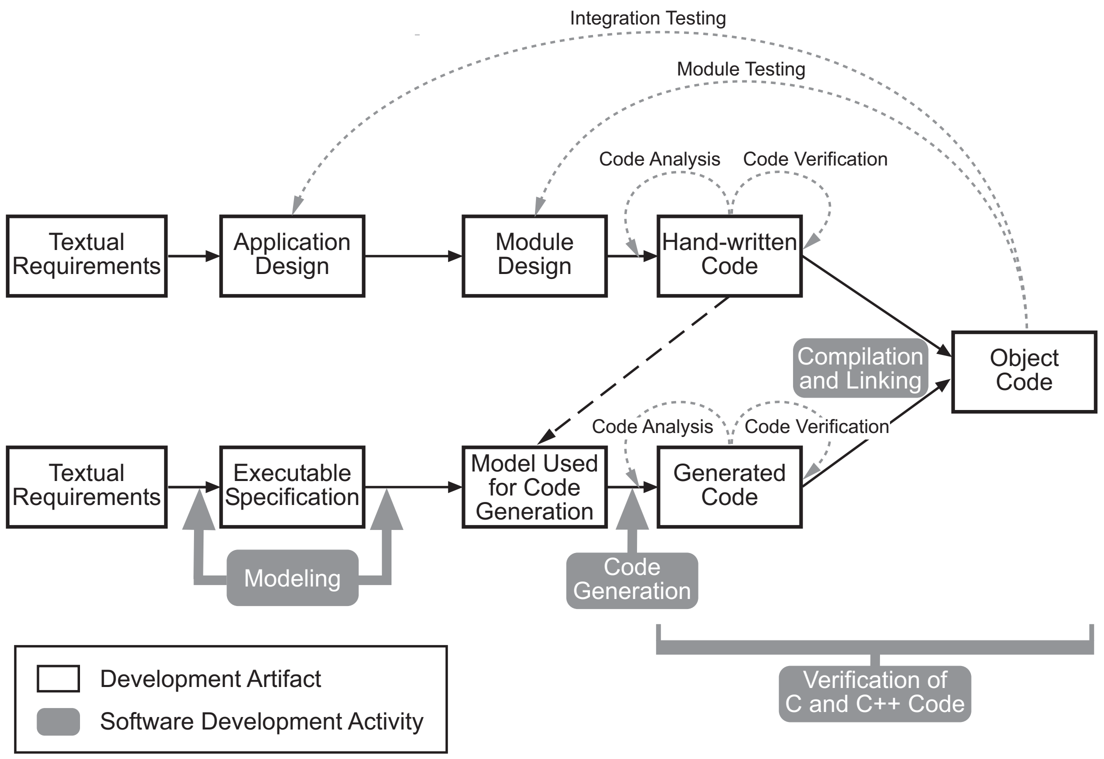
The goal of Polyspace verification is to improve the quality of the software by identifying implementation issues in the code, and proving that the code is both semantically and logically correct.

Polyspace verification allows you to find run-time errors:

- In hand-coded portions within the generated code
- In the model used for production code generation
- In the integration of manually written and generated code

Verification Workflow

The workflow is different for manually written code, generated code, and mixed code. Polyspace products can perform code verification as part of any of these workflows. The following figure shows a suggested verification workflow for manually written and mixed code.



Workflow for Verification of Generated and Mixed Code

Note Solid arrows in the figure indicate the progression of software development activities.

The verification workflow consists of the following steps:

1. The project leader configures a Polyspace project to meet defined quality goals.
2. Developers manually code sections of the application.
3. Developers or testers perform **Polyspace verification** of manually coded sections within the generated code, and review verification results according to the established quality goals.
4. Developers create Simulink model based on requirements.

5. Developers validate model to prove it is logically correct (using tools such as Simulink Model Advisor, and the Simulink Coverage™ and Simulink Design Verifier products).
6. Developers generate code from the model.
7. Developers or testers perform **Polyspace verification** on the entire software component, including both manually written and generated code.
8. Developers or testers review verification results according to the established quality goals.

Note Polyspace Code Prover allows you to quickly track issues identified by the verification back to the block in the Simulink model.

Costs and Benefits

Simulink Design Verifier verification can identify errors in textual designs or executable models that are not identified by other methods. The following table shows how errors in textual designs or executable models can appear in the resulting code.

Examples of Common Run-Time Errors

Type of Error	Design or Model Errors	Code Errors
Arithmetic errors	• Incorrect Scaling • Unknown calibrations • Untested data ranges	• Overflows/Underflows • Division by zero • Square root of negative numbers
Memory corruption	• Incorrect array specification in state machines • Incorrect legacy code (look-up tables)	• Out of bound array indexes • Pointer arithmetic
Data truncation	• Unexpected data flow	• Overflows/Underflows • Wrap-around
Logic errors	• Unreachable states • Incorrect Transitions	• Non initialized data • Dead code

Project Managers — Integrating Polyspace Verification with Configuration Management Tools

User Description

This workflow applies to project managers responsible for establishing check-in criteria for code at different development stages.

Quality

The goal of Polyspace verification is to test that code meets established quality criteria before being checked in at each development stage.

Verification Workflow

The verification workflow consists of the following steps:

1. Project manager defines quality goals, including individual quality levels for each stage of the development cycle.
2. Project leader configures a Polyspace project to meet quality goals.
3. Developers or testers run verification at the following stages:
   - Daily check-in — On the files currently under development. Compilation must complete without the permissive option.
   - Pre-unit test check-in — On the files currently under development.
   - Pre-integration test check-in — On the whole project, ensuring that compilation can complete without the permissive option. This stage differs from daily check-in because link errors are highlighted.
   - Pre-build for integration test check-in — On the whole project, with multitasking aspects accounted for as required.
   - Pre-peer review check-in — On the whole project, with multitasking aspects accounted for as required.
4. Developers or testers review verification results for each check-in activity to confirm the code meets the required quality level. For example, the transition criterion could be: “No defect found in 20 minutes of selective orange review”

Define Your Requirements

Before launching verification, define your requirements from the verification process. Defining your requirements helps decide which analysis options and results are relevant for you.

In this section...
"Define Broad Requirements for Verification" on page 2-19
"Define Specific Requirements for Verification" on page 2-20

Define Broad Requirements for Verification

This example shows how to define your broad requirements before you begin a Polyspace Code Prover verification, and then implement them in your verification process.

1. Prepare a set of quality levels for your application. A quality level chart can be like this:

   Software Quality Levels

Criteria	Software Quality Levels			
	QL1	QL2	QL3	QL4
Document static information	X	X	X	X
Enforce MISRA C coding rules in `SQO-subset1`	X	X	X	X
Review all red checks	X	X	X	X
Review all gray checks	X	X	X	X
Review critical orange checks		X	X	X
Review all orange checks			X	X
Enforce MISRA C coding rules in `SQO-subset2`			X	X
Analyze dataflow			X	X

2. Depending on the quality level that you want to implement, choose your verification options. The options appear on the **Configuration** pane in the Polyspace user interface.

For instance, if you want to implement level QL1, under **Coding Rules & Code Metrics**, for the option **Check MISRA C:2004**, select `SQO-subset1`.

3 Depending on the quality level that you want to implement, plan your review process for the verification results. Your review process involves options in the Polyspace interface.

For instance, if you want to implement level QL1, on the **Results List** pane, filter only red and gray checks.

Define Specific Requirements for Verification

This example shows how to define specific requirements before you begin a Polyspace Code Prover verification, and then implement them in your verification process.

Specify Code Constructs

1 Prepare a list of constructs that you want to retain in your code or remove from it.

2 On the **Configuration** pane, specify the verification options corresponding to your requirements.

For instance, you can have the following requirements and choose the corresponding options.

Requirement	Option
Detect overflows only on signed integer computations.	Under **Check Behavior**, for **Detect overflows**, select `signed`.
Allow a pointer to one structure field to point to another field of the same structure.	Under **Check Behavior**, select **Enable pointer arithmetic across fields**.
Do not allow global variables to be initialized by default.	Under **Inputs & Stubbing**, select **Ignore default initialization of global variables**.

Specify Coding Rules

1 Prepare a list of coding rules for your code.

2 On the **Configuration** pane, under the **Coding Rules & Code Metrics** node, specify your coding rules. For more information, see “Set Up Coding Rules Checking” on page 12-2.

Specify Results to Review

1. Prepare a list of files or list of checks that you want to review.
2. After you run your verification, apply appropriate filters to focus your review on those files or checks. For more information, see "Filter and Group Results" on page 8-113.

3

Setting up Project in User Interface

- “Create Project Automatically” on page 3-2
- “Requirements for Project Creation from Build Systems” on page 3-5
- “Compiler Not Supported for Project Creation from Build Systems” on page 3-8
- “Slow Build Process When Polyspace Traces the Build” on page 3-16
- “Check if Polyspace Supports Build Scripts” on page 3-17
- “Troubleshooting Project Creation from MinGW Build” on page 3-19
- “Create Project Manually” on page 3-20
- “Create Project Using Configuration Template” on page 3-25
- “Update Project” on page 3-29
- “Modularize Project Manually” on page 3-34
- “Modularize Project Automatically” on page 3-37
- “Create Project Using Visual Studio Information” on page 3-41
- “Troubleshooting Project Creation from Visual Studio Build” on page 3-43

Create Project Automatically

If you use build automation scripts to build your source code, you can automatically setup a Polyspace project from your scripts. The automatic project setup runs your automation scripts to determine:

- Source files.
- Includes.
- Target & compiler options. For more information on these options, see "Target & Compiler".

1. Select **File** > **New Project**.
2. On the Project – Properties dialog box, after specifying the project name and location, under **Project configuration**, select **Create from build command**.
3. On the next window, enter the following information:

Field	Description
Specify command used for building your source files	If you use an IDE such as Visual Studio® or Eclipse™ to build your project, specify the full path to your IDE executable or navigate to it using the button. For a tutorial using Visual Studio, see "Create Project Using Visual Studio Information" on page 3-41. **Example:** `"C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\VCExpress.exe"` If you use command-line tools to build your project, specify the appropriate command. **Example:** • `make -B -f` *`makefileName`* or `make -W` *`makefileName`* • `"mingw32-make.exe -B -f makefilename"`

Field	Description
Specify working directory for running build command	Specify the folder from which you run your build automation script. If you specify the full path to your executable in the previous field, this field is redundant. Specify any folder.
Add advanced configure options	Specify additional options for advanced tasks such as incremental build. For the full list of options, see the `-options value` argument for `polyspaceConfigure`.

4 Click Run.

- If you entered your build command, Polyspace runs the command and sets up a project.
- If you entered the path to an executable, the executable runs. Build your source code and close the executable. Polyspace traces your build and sets up a project.

  For example, in Visual Studio 2010, use **Tools** > **Rebuild Solution** to build your source code. Then close Visual Studio.

If a failure occurs, see if your build command meets the requirements for automatic project setup. In some cases, you can modify your build command to work around the limitations. For more information, see "Requirements for Project Creation from Build Systems" on page 3-5.

5 Click **Finish**.

The new project appears on the **Project Browser** pane. To close the project at any time, in the **Project Browser**, right-click the project node and select **Close**.

6 If you updated your build command, you can recreate the Polyspace project from the updated command. To recreate an existing project, on the **Project Browser**, right-click the project name and select **Update Project**.

Note

- In the Polyspace interface, it is possible that the created project will not show implicit defines or includes. The configuration tool does include them. However, they are not visible.

- By default, Polyspace assigns the latest version of the compiler to your project. If you have compilation errors in your project, check the setting for `Compiler (-compiler)`. If it does not apply to you, change it to a more appropriate one.

  For instance, if the compiler setting is `visual12` but you are using Microsoft® Visual C++® 2010, change the setting to `visual10`.

  For an example, see "Compilation Error After Creating Project from Visual Studio Build" on page 3-43.
- If your build process requires user interaction, you cannot run the build command from the Polyspace user interface and do an automatic project setup.

See Also

Related Examples

- "Create Project Using Visual Studio Information" on page 3-41
- "Create Project Manually" on page 3-20
- "Update Project" on page 3-29

More About

- "Compiler Not Supported for Project Creation from Build Systems" on page 3-8
- "Slow Build Process When Polyspace Traces the Build" on page 3-16
- "Check if Polyspace Supports Build Scripts" on page 3-17

Requirements for Project Creation from Build Systems

For automatic project creation from build systems, your build commands or makefiles must meet certain requirements.

For more information on automatic project creation, see:

- “Create Project Automatically” on page 3-2
- “Create Project Automatically at Command Line” on page 6-17
- “Create Project Automatically from MATLAB Command Line” on page 6-35

The requirements for your build command are as follows:

- 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:

  - Visual C++ compiler
  - `gcc`
  - `clang`
  - `MinGW` compiler
  - `IAR` compiler

  If your compiler configuration is not available to Polyspace:

  - Write a compiler configuration file for your compiler in a specific format. For more information, see “Compiler Not Supported for Project Creation from Build Systems” on page 3-8.
  - Contact MathWorks Technical Support. For more information, see “Contact Technical Support” on page 7-21.

- 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 3-17.
- 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.
- You cannot trace your build command on the operating system OS X El Capitan if the security feature System Integrity Protection (SIP) is active. Before tracing your build command, disable this feature. You can reenable this feature after tracing the build command.
- If your computer hibernates during the build process, Polyspace might not be able to trace your build.
- With the TASKING compiler, if you use an alternative sfr file with extension `.asfr`, Polyspace might not be able to locate your file. If you encounter an error, explicitly

`#include` your `.asfr` file in the preprocessed code using the option `Include (-include)`.

Typically, you use the statement `#include __SFRFILE__(__CPU__)` along with the compiler option `--alternative-sfr-file` to specify an alternative sfr file. The path to the file is typically `*Tasking_C166_INSTALL_DIR*\include\sfr\reg*CPUNAME*.asfr`. For instance, if your TASKING compiler is installed in `C:\Program Files\Tasking\C166-VX_v4.0r1\` and you use the CPU-related flag `–Cxc2287m_104f` or `--cpu=xc2287m_104f`, the path is `C:\Program Files\Tasking\C166-VX_v4.0r1\include\sfr\regxc2287m.asfr`.

Note Your environment variables are preserved when Polyspace traces your build command.

See Also

`polyspaceConfigure`

Related Examples

- “Create Project Automatically” on page 3-2
- “Create Project Automatically at Command Line” on page 6-17
- “Create Project Automatically from MATLAB Command Line” on page 6-35

More About

- “Slow Build Process When Polyspace Traces the Build” on page 3-16

Compiler Not Supported for Project Creation from Build Systems

Issue

Your compiler is not supported for automatic project creation from build commands.

For more information on automatic project creation, see:

- “Create Project Automatically” on page 3-2
- “Create Project Automatically at Command Line” on page 6-17
- “Create Project Automatically from MATLAB Command Line” on page 6-35

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 3-5.

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 `matlabroot\polyspace\configure\compiler_configuration\`.
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 `matlabroot\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.

The following table lists the XML elements in the file with a description of what the content within the element represents.

XML Element	Content Description	Content Example for GNU C Compiler

`<compiler_names><name> ...`
 	
`</name><compiler_names>`
	Name of the compiler executable. This executable transforms your `.c` files into object files. You can add several binary names, each in a separate `<name>...</name>` element. The software checks for each of the provided names and uses the compiler name for which it finds a match.
	
	You must not specify the linker binary inside the `<name>...</name>` elements.
	
	If the name that you specify is present in an existing compiler configuration file, an error occurs. To avoid the error, use the additional option `-compiler-config` *`my_compiler`*`.xml` when tracing the build so that the software explicitly uses your compiler configuration file.
		• `gcc`
		• `gpp`

XML Element	Content Description	Content Example for GNU C Compiler
<include_options><opt> ... </opt></include_options>	The option that you use with your compiler to specify include folders. To specify options where the argument immediately follows the option, use an isPrefix attribute for opt and set it to true.	-I
<system_include_options> <opt> ... </opt> </system_include_options>	The option that you use with your compiler to specify system headers. To specify options where the argument immediately follows the option, use an isPrefix attribute for opt and set it to true.	-isystem
<preinclude_options><opt> ... </opt></preinclude_options>	The option that you use with your compiler to force inclusion of a file in the compiled object. To specify options where the argument immediately follows the option, use an isPrefix attribute for opt and set it to true.	-include

XML Element	Content Description	Content Example for GNU C Compiler

`<define_options><opt> ...`
 	
`</opt></define_options>`
	The option that you use with your compiler to predefine a macro.
	
	To specify options where the argument immediately follows the option, use an `isPrefix` attribute for `opt` and set it to `true`.
		`-D`

`<undefine_options><opt> ...`
 	
`</opt></undefine_options>`
	The option that you use with your compiler to undo any previous definition of a macro.
	
	To specify options where the argument immediately follows the option, use an `isPrefix` attribute for `opt` and set it to `true`.
		`-U`

XML Element	Content Description	Content Example for GNU C Compiler
<semantic_options><opt> ... </opt></semantic_options>	The options that you use to modify the compiler behavior. These options specify the language settings to which the code must conform. You can use the isPrefix attribute to specify multiple options that have the same prefix and the numArgs attribute to specify options with multiple arguments. For instance: • Instead of <opt>-m32</opt> <opt>-m64</opt> You can write <opt isPrefix="true">-m</opt>. • Instead of <opt>-std=c90</opt> <opt>-std=c99</opt> You can write <opt numArgs="1">-std</opt>. If your makefile uses -std c90 instead of -std=c90, this notation also supports that usage.	• -ansi • -std =C90 • -std =c++11 • -fun signed -char

XML Element	Content Description	Content Example for GNU C Compiler
`<dialect> ... </dialect>`	The Polyspace dialect that corresponds to or closely matches your compiler dialect. The content of this element directly translates to the option **Dialect** in your Polyspace project or options file. For the complete list of dialects, on the **Configuration** pane, select **Target & Compiler**.	`gnu4.7`

`<preprocess_options_list>`
 	
`<opt> ... </opt>`
 	
`</preprocess_options_list>`
	The options that specify how your compiler generates a preprocessed file.
	
	You can use the macro `$(OUTPUT_FILE)` if your compiler does not allow sending the preprocessed file to the standard output. Instead it defines the preprocessed file internally.
		`-E`
		
		For an example of the `$(OUTPUT_FILE)` macro, see the existing compiler configuration file `cl2000.xml`.

XML Element	Content Description	Content Example for GNU C Compiler
`<preprocessed_output_file> ... </preprocessed_output_file>`	The name of file where the preprocessed output is stored. You can use the following macros when the name of the preprocessed output file is adapted from the source file: • `$(SOURCE_FILE)`: Source file name • `$(SOURCE_FILE_EXT)`: Source file extension • `$(SOURCE_FILE_NO_EXT)`: Source file name without extension For instance, use `$(SOURCE_FILE_NO_EXT).pre` when the preprocessor file name has the same name as the source file, but with extension `.pre`.	For an example of this element, see the existing compiler configuration file `xc8.xml`.

`<src_extensions><ext> ...`
 	
`</ext></src_extensions>`
	The file extensions for source files.	• `c`
		• `cpp`
		• `c++`

`<obj_extensions><ext> ...`
 	
`</ext></obj_extensions>`
	The file extensions for object files.	

`<precompiled_header_extensions> ...`
 	
`</precompiled_header_extensions>`
	The file extensions for precompiled headers (if available).	

XML Element	Content Description	Content Example for GNU C Compiler

`<polyspace_c_extra_options_list>`
 	
`<opt> ... </opt>`
 	
`</polyspace_c_extra_options_list>`
	Additional options that will be added to your project configuration	To avoid compilation errors due to non-ANSI® extension keywords, enter `-D` *keyword*. For more information, see `Preprocessor definitions (-D)`.

`<polyspace_cpp_extra_options_list>`
 	
`<opt> ... </opt>`
 	
`</polyspace_cpp_extra_options_list>`
	Additional options that will be added to your C++ project configuration	To avoid compilation errors due to non-ANSI extension keywords, enter `-D` *keyword*. For more information, see `Preprocessor definitions (-D)`.

4 After saving the edited XML file to *matlabroot*\polyspace\configure \compiler_configuration\, create a project automatically using your build command.

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.

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**. For more information, see “Create Project Automatically” on page 3-2.
- If you trace your build from the DOS, UNIX or MATLAB® command line, use this flag with the `polyspace-configure` command or `polyspaceConfigure` function.

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.

For more information on automatic project creation from build commands, see:

- “Create Project Automatically” on page 3-2
- “Create Project Automatically at Command Line” on page 6-17
- “Create Project Automatically from MATLAB Command Line” on page 6-35

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-nodesktop` 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__))`

Create Project Manually

To analyze your sources files in the user interface, you must create a Polyspace project. The project consists of your source files, include folders and one or more modules. You add all or some of your source files to a module, change the default analysis options if you want, and run analysis on the module.

If you do not use build automation scripts to build your source code, you can create a Polyspace project manually.

Otherwise, see "Create Project Automatically" on page 3-2. If automatic project creation is not supported for your compiler, see the suggestions in "Requirements for Project Creation from Build Systems" on page 3-5. If the suggestions do not work, create a project manually.

Tip In the Polyspace user interface, you can quickly change to an arrangement of panes dedicated to project setup. Select **Window** > **Reset Layout** > **Project Setup**.

Create Project

When creating a new project, you must know:

- Location of source files
- Location of include files
- Location where you want to store analysis results

1. Select **File** > **New Project**.
2. In the Project – Properties window, specify properties for your project:

   - **Project name**
   - **Location**: Folder where you will store the project file (`.psprj` file) and the results unless you specify otherwise. You can use the `.psprj` file to reopen the project.

     The software assigns a default location to your project called your Polyspace Workspace. You can change this default in the Polyspace Preferences on the **Project and Results Folder** tab.

- Clear the **Use template** check box unless you have a template you want to use.

3 On the next screen, add source folders to your project.

   a Use the **Browse** button to navigate to the folder containing the source files you want to analyze.

      By default, Polyspace looks for `.c`, `.cpp`, `.cxx`, or `.cc` files. If you use other file extensions, before closing the dialog box, change the **Files of types** option.

   b If you chose a source folder with subfolders but do not want to analyze files in the subfolders, clear the check box **Add recursively**.

   c (Linux only) Often, compilers add symbolic links in your source folders during compilation. If your folder contains symbolic links to other folders but you do not want to add source files from the other folders, select **Exclude symbolic links**.

   d Click **Add Source Folder**. All source files found under this folder are added to your Polyspace project.

Tip To see the full path of your files, toggle the button.

---

   e If you do not want to analyze all the files under your source folder, 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**.

4 On the next screen, add include folders to your project. The analysis looks for include files relative to the 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`.

   a Use the **Browse** button to navigate to your folder containing the include files needed for compilation.

      By default, Polyspace looks for `.h`, `.hpp`, or `.hxx` files. If you use other file extensions, before closing the dialog box, change the **Files of types** option.

   b If you chose an include folder that contains subfolder and you want to add those include folders as well, select the check box **Include all subfolders**.

c (Linux only) Often, compilers add symbolic links in your folders during compilation. If your folder contains symbolic links to other folders but you do not want to add includes from the other folders, select **Exclude symbolic links**.

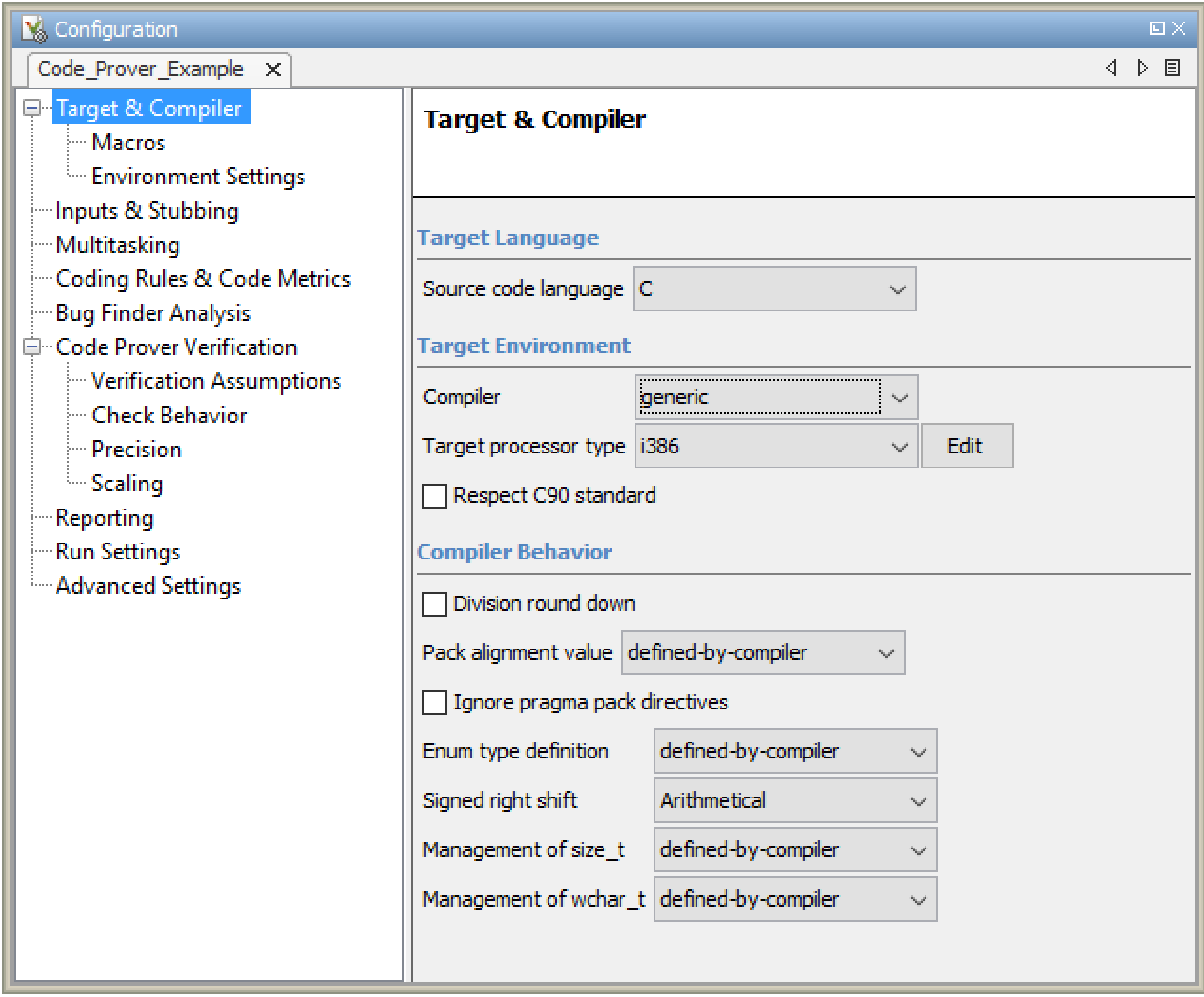
d Click **Add Include Folders**. The include folder is added to your Polyspace project.

Specify Analysis Options

You can either retain the default analysis options used by the software or change them to your requirements.

Each project consists of one or more modules. Before running verification on a module, you can change the analysis options. Each module has a **Configuration** that consists of the default analysis options. To change the analysis options:

1. On the **Project Browser**, below the **Configuration** node of the module, select the configuration.
2. Change the options on the **Configuration** pane.



For instance:

- To specify the target processor, select the **Target & Compiler** node. Select your processor from the **Target processor type** drop-down list.
- To specify verification precision, select the **Precision** node. Select a number for **Precision level**.

Using the command-line names in the **Advanced Settings** node in the user interface, you can specify analysis options multiple times. This flexibility allows you to customize pre-made configurations without having to remove options.

If you specify an option multiple times, only the last setting is used. For example, in the user interface, on the **Target & Compiler** node, you can specify the target as **c18** and on the **Advanced Settings** node, enter `-target i386`. These two targets count as multiple analysis option specifications. Polyspace uses the target specified in the **Advanced Settings** dialog box, `i386`.

You can also create another configuration in your module. For more information, see "Create Configurations in Module" on page 3-35.

For more information on the options, see "Analysis Options".

See Also

Related Examples

- "Create Project Automatically" on page 3-2
- "Create Project Using Configuration Template" on page 3-25
- "Update Project" on page 3-29

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 Code Prover 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.

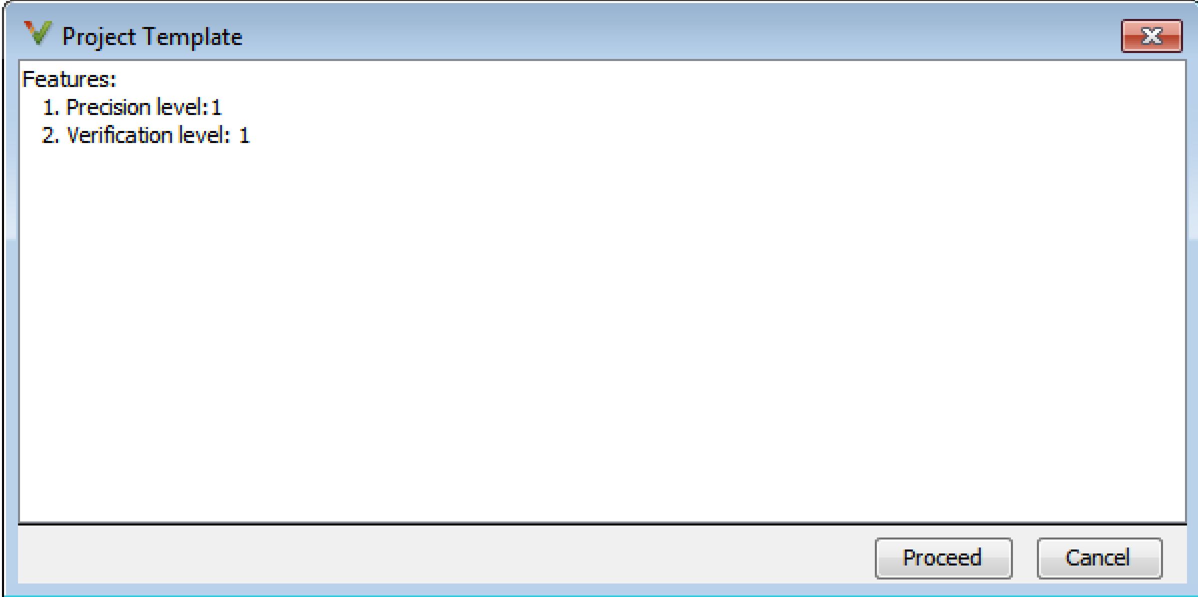
   If your compiler does not appear in the list of predefined templates, select **Baseline_C** or **Baseline_C++**.
4. On the next screen, add your source files and include folders. For more information, see “Create Project Manually” on page 3-20.

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.

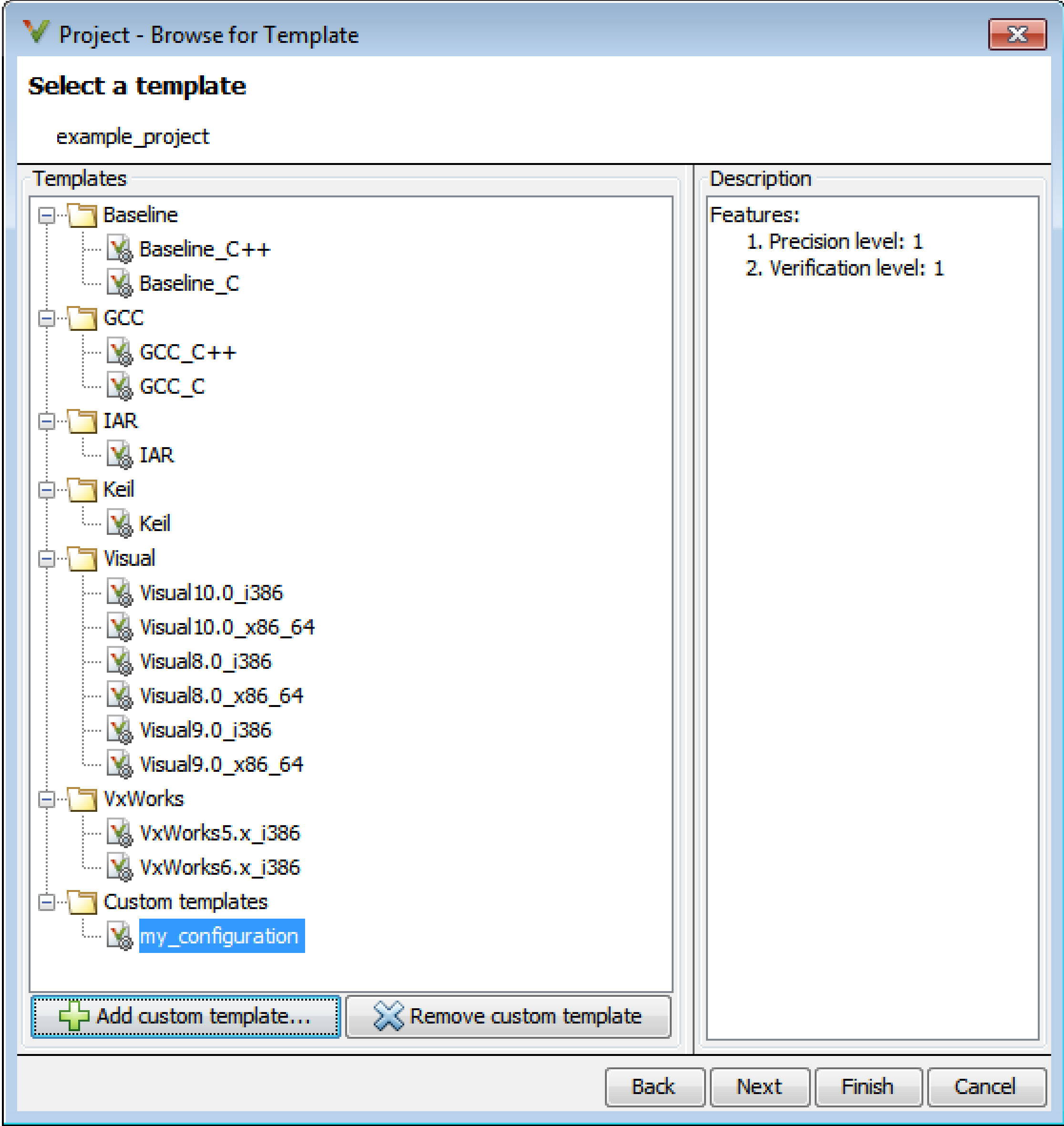
- To create a template from a project that is open on the **Project Browser** pane:

  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.



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

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


Project - Browse for Template
Select a template
example_project
Templates
Baseline
Baseline_C++
Baseline_C
GCC
GCC_C++
GCC_C
IAR
IAR
Keil
Keil
Visual
Visual10.0_i386
Visual10.0_x86_64
Visual8.0_i386
Visual8.0_x86_64
Visual9.0_i386
Visual9.0_x86_64
VxWorks
VxWorks5.x_i386
VxWorks6.x_i386
Custom templates
my_configuration
Add custom template...
Remove custom template
Description
Features:
1. Precision level: 1
2. Verification level: 1
Back
Next
Finish
Cancel

See Also

Related Examples

- "Create Project Automatically" on page 3-2
- "Create Project Manually" on page 3-20
- "Update Project" on page 3-29

Update Project

In this section...
“Change Folder Path” on page 3-29
“Refresh Source List” on page 3-29
“Refresh Project Created from Build Command” on page 3-30
“Add Source and Include Folders” on page 3-30
“Manage Include File Sequence” on page 3-32
“Change Analysis Options” on page 3-32

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.

1. In the **Project Browser**, right-click the top sources folder and select **Modify Path**.
2. In the dialog box, in the text box, change the path to the new location.
3. Click **Save Changes**.
4. Click **Finish**.
5. 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.

1. 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:

1. On the **Project Browser** pane, right-click the project folder and select **Update Project**.
2. Enter the same information you did when creating the original project. For more information, see "Create Project Automatically" on page 3-2.

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:

1. 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, follow these steps:

1. In the **Project Browser**, right-click your project or the **Source** or **Include** folder in your project.
2. Select **Add Source Folder** or **Add Include Folder**.
3. Add source folders to your project:

   a. Use the **Browse** button to navigate to the folder containing the source files you want to analyze.

      By default, Polyspace looks for `.c`, `.cpp`, `.cxx`, or `.cc` files. If you use other file extensions, before closing the dialog box, change the **Files of types** option.

   b. If you chose a source folder that contains subfolder and you do not want to analyze source files in those subfolders, clear the check box **Add recursively**.

   c. (Linux only) Often, compilers add symbolic links in your source folders during compilation. If your folder contains symbolic links to other folders but you do not want to add source files from the other folders, select **Exclude symbolic links**.

d Click **Add Source Folder**. All source files found under the folder are added to your Polyspace project.

Tip To see the full path of your files, click the button.

---

e If you do not want to analyze all the files under your source folder, 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**.

Repeat these steps as many times as necessary, then click **Next**.

4 Add include folders to your project.

a Use the **Browse** button to navigate to your folder containing the include files needed for compilation.

By default, Polyspace looks for `.h`, `.hpp`, or `.hxx` files. If you use other file extensions, before closing the dialog box, change the **Files of types** option.

b If you chose an include folder that contains subfolder and you want to add those include folders as well, select the check box **Include all subfolders**.

c (Linux only) Often, compilers add symbolic links in your folders during compilation. If your folder contains symbolic links to other folders but you do not want to add includes from the other folders, select **Exclude symbolic links**.

d Click **Add Include Folders**. The include folder is added to your Polyspace project.

Repeat these steps as many times as necessary, then click **Finish**. The new project opens in the **Project Browser** pane. Your source files are copied automatically to the first module in the project.

5 Click **Finish**.

6 Before running an analysis, you must copy the source files to a module.

a Select the source files that you want to copy. To select multiple files together, press the **Ctrl** key while selecting the files.

b Right-click your selection.

c 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`.

Include
H:\Polyspace\Sources\Manage_Include_File_Sequence\Folder_2
H:\Polyspace\Sources\Manage_Include_File_Sequence\Folder_1

To change the order of include folders:

1. In your project, expand the **Include** folder.
2. Select the include folder or folders that you want to move.
3. To move the folder, click either or .

Change Analysis Options

For later verifications, you might have to change your analysis options. For instance:

- To avoid compilation errors in Polyspace for constructs that are allowed by your compiler, specify your target and compiler options.

  For more information, see “Target & Compiler”.
- If you provide partially developed code, you can specify external constraints to stand in for the remaining code. Towards the end of your development cycle, as you provide more complete code for verification, you can remove some of these constraints.

  For more information, see “Inputs & Stubbing”.
- If your code is intended for multitasking, you can specify your entry points and protection mechanisms.

For more information, see “Multitasking”.

- To allow Polyspace to prove more operations and therefore produce fewer non-critical orange checks, you can specify appropriate options.

  For more information, see “Reduce Orange Checks” on page 10-16.

For more information, see “Specify Analysis Options” on page 3-22.

See Also

Related Examples

- “Create Project Automatically” on page 3-2
- “Create Project Manually” on page 3-20

Modularize Project Manually

You can create multiple modules in a Polyspace Code Prover project. In each module, you can copy all or some of your source files.

On the **Project Browser** pane, each module contains the following nodes.

Node	Content
Source	All or some of the source files in the project. When you run verification on the module, the software verifies these source files.
Configuration	One or more configurations. Each configuration consists of a set of analysis options.
Result	One or more results.

In your file system, each module corresponds to a subfolder of your project folder.

Note If you add your source files when creating a new project, they are automatically copied to the first module, **Module_1**. If you add them later, you must copy them manually to a module.

In this section...
"Create New Module" on page 3-34
"Create Configurations in Module" on page 3-35

Create New Module

Suppose you have one module, **Module_1**, in your project.

1 Do one of the following on the **Project Browser** pane:

- Select your project. Click the button on the **Project Browser** toolbar.
- Right-click your project or the existing module. Select **Create New Module**.

You see a new module, **Module_2**, in your project. To rename the module, right-click the module name.

2 In your project, below the **Source** node, right-click the files that you want to add to the module. From the context menu, select **Copy to** > **Module_2**.

   The software displays these files below the **Source** node of `Module_2`.

Create Configurations in Module

By default, when you create a new module, it contains a configuration with the default analysis options. To run verification on the module with different options, do one of the following:

- Change the analysis options in this configuration.
- Create a new configuration and change the options in the new configuration. You can retain the default analysis options in the original configuration.

Tip To copy a configuration to another module, right-click the configuration. Select **Copy Configuration to** > ***`Module_name`***.

To create a new configuration in your module:

1 Right-click the **Configuration** folder in the module. From the context menu, select **Create New Configuration**.

   - On the **Project Browser** pane, the software displays a new configuration *`project_name_1`*. To rename the configuration, right-click the configuration and select **Rename Configuration**.
   - On the **Configuration** pane, the new configuration appears as an additional tab.

2 On the **Configuration** pane, specify the analysis options for the new configuration.

3 To use this new configuration, double-click it.

   When you run a new verification on the module, it uses the analysis options in this configuration.

4 To see the configuration you used for a certain result, right-click the result on the **Project Browser**. Select **Open Configuration**.

   You can see a read-only form of the configuration.

Note If you are viewing the results and do not have the corresponding project open on your **Project Browser**, to see the configuration you used, select the link **View configuration for results** on the **Dashboard** pane.

See Also

Related Examples

- "Modularize Project Automatically" on page 3-37

Modularize Project Automatically

If the source code in your project represents a single application, you might want to analyze all of the code together. However, if the application is extremely large, the verification can take a long time, for example, days.

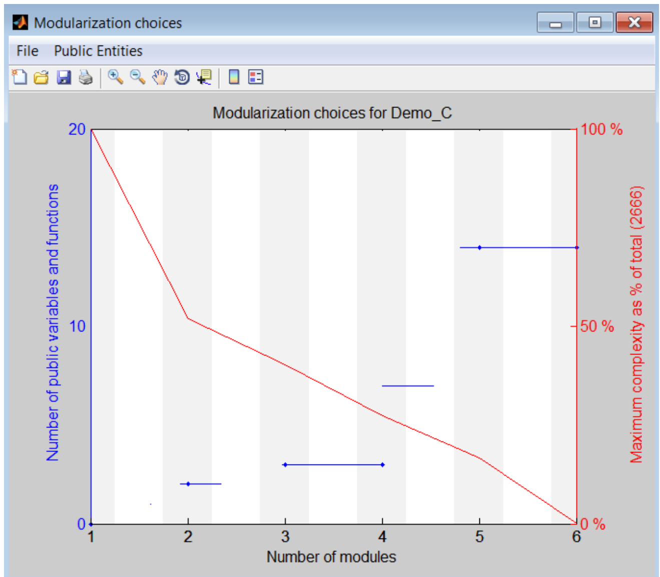
For a large application, Polyspace allows you to:

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

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

To partition your application into modules:

1. Run an initial verification, which performs a limited analysis but processes all the files of your application. For example, run a verification with the following **Precision** pane settings:
   - **Precision level** — `0`
   - **Verification level** — `Software Safety Analysis level 0`
2. In the **Project Browser** view, select the results folder.
3. Select **Tools** > **Run Modularize**. The software analyzes your application code and displays two plots in a new Modularization choices window.



The plots show the following information:

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

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

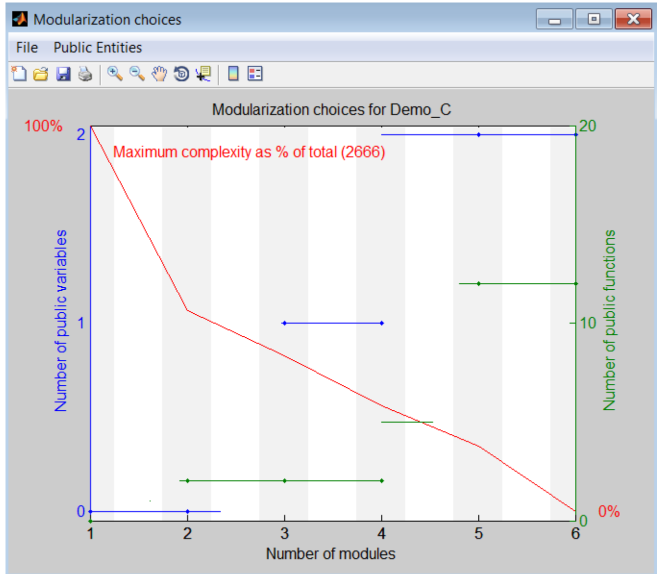
The number of partitioned modules that you choose involves a trade-off between the following:

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

Select a number just after a big drop in maximum complexity and before a big increase in the number of public functions and variables. The precision of a modular

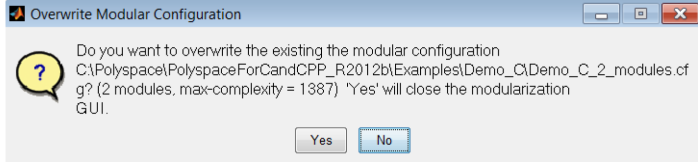
verification can be very sensitive to the number of public variables. If the series of horizontal blue lines ascends so gradually that there is no clear number choice, then:

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



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

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



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

You can now verify each module separately.

See Also

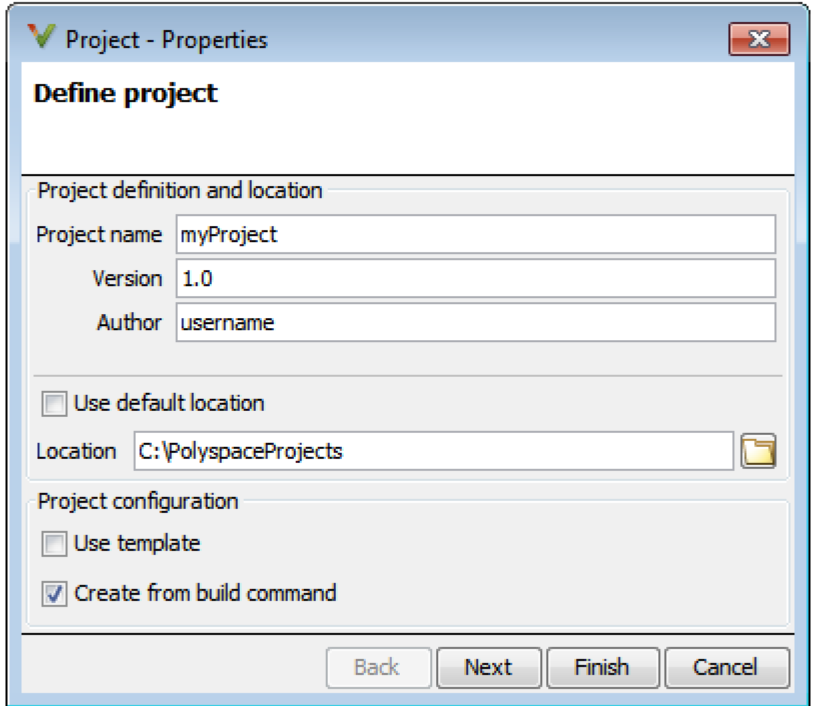
Related Examples

- "Modularize Project Manually" on page 3-34

Create Project Using Visual Studio Information

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

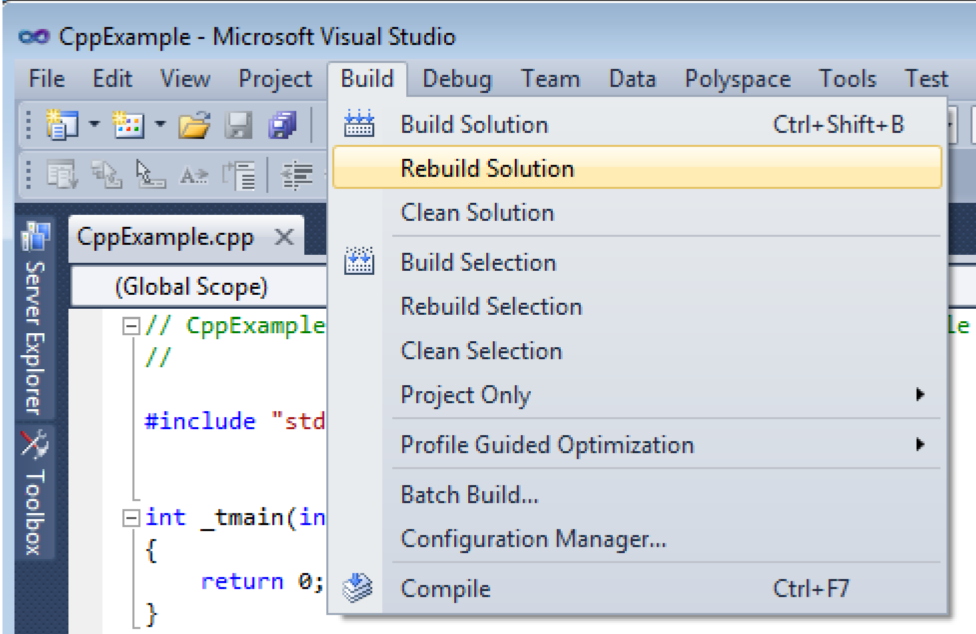
1. In the Polyspace interface, select **File** > **New Project**.
2. In the Project – Properties window, under **Project Configuration**, select **Create from build command** and click **Next**.



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\VCExpress.exe`".
4. In the field **Specify working directory for running build command**, enter `C:\`. Click Run.

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

If you already have a Visual Studio project, open the existing project and build a clean solution. 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 3-43

Troubleshooting Project Creation from Visual Studio Build

In this section...
"Cannot Create Project from Visual Studio Build" on page 3-43 "Compilation Error After Creating Project from Visual Studio Build" on page 3-43

Cannot Create Project 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
   ```

Compilation Error After Creating Project from Visual Studio Build

Issue

After you automatically set up your project from a Visual Studio 2010 build, you face compilation errors.

Possible Cause

By default, Polyspace assigns the latest version of the compiler, `visual12.0` to your project. This assignment can cause compilation errors. For more information on the option to specify compilers, see `Compiler (-compiler)`.

Solution

To avoid the errors, do one of the following:

- After automatic project setup:

  1. Open the project in the user interface. On the **Configuration** pane, select **Target & Compiler**.

2. Check the setting for **Compiler**. If it is set to `visual12.0`, change it to `visual10`.

Note If you are creating an options file from your Visual Studio 2010 build, check the `-compiler` argument. If it is set to `visual12.0`, change it to `visual10`.

- Before automatic project setup:

1. Open the file `cl.xml` in *`matlabroot`*`\polyspace\configure\compiler_configuration\` where *`matlabroot`* is your MATLAB installation folder such as `C:\Program Files\R2015a`.
2. Change the line

   ```
   <dialect>visual12.0</dialect>
   ```

   to

   ```
   <dialect>visual10</dialect>
   ```
3. Create your project or options file. The compiler is already assigned to `visual10`.

Setting Up Polyspace User Interface

- "Organize Layout of Polyspace User Interface" on page 4-2
- "Specify External Text Editor" on page 4-4
- "Change Default Font Size" on page 4-6
- "Customize Results Folder Location and Name" on page 4-7
- "Storage of Polyspace Preferences" on page 4-8

Organize Layout of Polyspace User Interface

The Polyspace user interface has two default layouts of panes.

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

Project Browser	Configuration
	Output Summary

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

Results List	Result Details
	Dashboard

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

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

Create Your Own Layout

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

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

To move a pane to another location:

1 Float the pane in one of three ways:

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

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

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

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

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

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

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

Save and Reset Layout

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

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

Specify External Text Editor

This example shows how to 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** pane. If you prefer editing your source files in an external editor, you can change this default behavior.

1. Select **Tools** > **Preferences**.
2. On the Polyspace Preferences dialog box, select the **Editors** tab.
3. From the **Text editor** drop-down list, select **External**.
4. In the **Text editor** field, specify the path to your text editor. For example:

   ```
   C:\Program Files\Windows NT\Accessories\wordpad.exe
   ```
5. 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 the following editors:

   - `Emacs`
   - `Notepad++` — Windows only
   - `UltraEdit`
   - `VisualStudio`
   - `WordPad` — Windows only
   - `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`:

   a. In the **Text Editor** field, enter `/usr/bin/xterm`.

b From the **Arguments** drop-down list, select `Custom`.

c In the field to the right, enter `-e /usr/bin/vi $FILE`.

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

Change Default Font Size

This example shows how to change the default font size in the Polyspace user interface.

1 Select **Tools** > **Preferences**.
2 On 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**.
3 Click **OK**.

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

Customize Results Folder Location and Name

By default, the software saves results in `Module_#` subfolders within the project folder. However, through the Polyspace Preferences dialog box, you can define a parent folder for your results:

1. Select **Tools** > **Preferences**.
2. On the **Project and Results Folder** tab, select the **Create new result folder** check box.
3. In the **Parent results folder location** field, specify the location that you want.

   Note If you do not specify a parent results folder, the software uses the active module folder as the parent folder.

4. If you want your results to be stored in a subfolder instead of directly in the parent folder, select **Add a subfolder using the project name**. This subfolder takes the name of the project.
5. If required, specify additional formatting options for the folder name . The options allow you to incorporate the following information into the name of the results folder:

   - **Result folder prefix** — A string that you define. Default is `Result`.
   - **Project variable** — Project, module, and configuration.
   - **Date format** — Date of analysis
   - **Time format** — Time of analysis
   - **Counter** — Count value that automatically increments by one for each new results folder

The software now creates a new results folders with the file name *`ResultFolderPrefix_ProjectVariable_DateFormat_TimeFormat_Counter`*.

Storage of Polyspace Preferences

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 \AppData\Roaming\MathWorks\MATLAB \polyspace_shared \polyspace_products.prf
- Linux : /home/*$User*/.matlab/polyspace_shared/polyspace_products.prf

Emulating Your Runtime Environment

- "Specify Target Environment and Compiler Behavior" on page 5-2
- "Provide Standard Library Headers for Polyspace Analysis" on page 5-6
- "Language Extensions Supported by Default" on page 5-8
- "Supported Keil or IAR Language Extensions" on page 5-10
- "Supported C++ 2011 Language Extensions" on page 5-12
- "Remove or Replace Keywords Before Compilation" on page 5-15
- "Gather Compilation Options Efficiently" on page 5-18
- "Verify C Application Without main Function" on page 5-20
- "Verify C++ Classes" on page 5-24
- "Specify External Constraints" on page 5-35
- "Constrain Global Variable Range" on page 5-40
- "Constrain Function Inputs" on page 5-42
- "Constrain Stubbed Functions" on page 5-44
- "Constraints" on page 5-46
- "XML File Format for Constraints" on page 5-56
- "Provide Context for C Code Verification" on page 5-65
- "Provide Context for C++ Code Verification" on page 5-67
- "Verify Multitasking Applications" on page 5-69
- "Manually Model Tasks if main Contains Infinite Loop" on page 5-72
- "Manually Model Scheduling of Tasks" on page 5-76
- "Manually Protect Shared Variables from Concurrent Access" on page 5-80

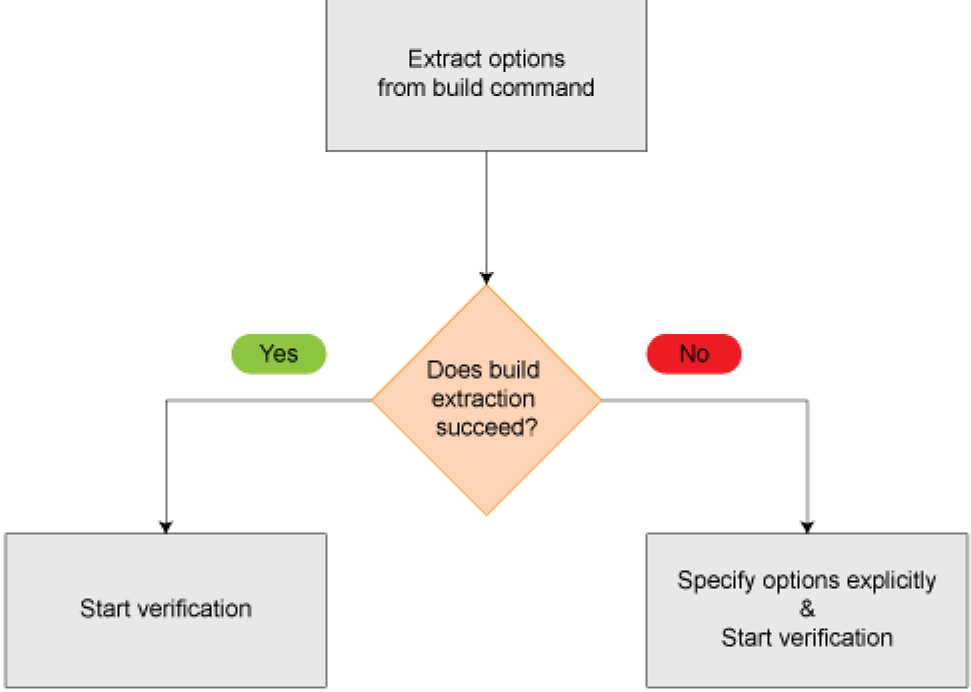
Specify Target Environment and Compiler Behavior

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.

For information on how to trace your build command from the:

- Polyspace user interface, see “Create Project Automatically” on page 3-2.
- DOS or UNIX command line, see “Create Project Automatically at Command Line” on page 6-17.
- MATLAB command line, see “Create Project Automatically from MATLAB Command Line” on page 6-35.

For Polyspace project creation, your build automation script (makefile) must meet certain requirements. See “Requirements for Project Creation from Build Systems” on page 3-5.

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 Polyspace user interface, 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-code-prover-nodesktop` command.
- At the MATLAB command line, specify arguments with the `polyspaceCodeProver` 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++11 extensions (-cpp11-extension)`: Select this option if you use C++11 extensions. See also "Supported C++ 2011 Language Extensions" on page 5-12.
  - `Respect C90 standard (-no-language-extensions)`: Select this option if you prefer that the verification use the C90 Standard (ISO/IEC 9899:1990). Otherwise, the verification uses the ANSI C99 Standard (ISO®/IEC 9899:1999) for compilation and checking of certain coding rules.

- 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 & 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 & Compiler".

- Compiler header files:

  If you specify the `diab`, `tasking` or `greenhills` compiler, you must specify the path to your compiler header files.

  - In the user interface, add the folder containing your compiler headers to the project.

For more information, see “Add Source and Include Folders” on page 3-30.

- At the command line, use the flag `-I` with the `polyspace-code-prover-nodesktop` command.

  For more information, see `-I`.

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 “Macros”.
- *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 “Errors from Conflicts with Polyspace Header Files” on page 7-56.

See Also

More About

- “Language Extensions Supported by Default” on page 5-8
- “Supported Keil or IAR Language Extensions” on page 5-10
- “Supported C++ 2011 Language Extensions” on page 5-12

Provide Standard Library Headers for Polyspace Analysis

Before Polyspace analyzes the code for bugs and run-time errors, it compiles your code. Even if the code compiles with your compiler, you can see compilation errors with Polyspace. If the error comes from a standard library function, it usually indicates that Polyspace is not using your compiler headers. To work around the errors, provide the path to your compiler headers.

This topic shows how to locate the standard library headers from your compiler. The code examples cause a compilation error that shows the location of the headers.

- To locate the folder containing your C compiler system headers, compile this C code by using your compilation toolchain:

  ```
  float fopen(float f);
  #include <stdio.h>
  ```

  The code does not compile because the `fopen` declaration conflicts with the declaration inside `stdio.h`. The compilation error shows the location of your compiler implementation of `stdio.h`. Your C standard library headers are all likely to be in that folder.
- To locate the folder containing your C++ compiler system headers, compile this C++ code by using your compilation toolchain:

  ```
  namespace std {
      float cin;
  }
  #include <iostream>
  ```

  The code does not compile because the `cin` declaration conflicts with the declaration inside `iostream.h`. The compilation error shows the location of your compiler implementation of `iostream.h`. Your C++ standard library headers are all likely to be in that folder.

After you locate the path to your compiler's header files, specify the path for the Polyspace analysis. For C++ code, specify the paths to both your C and C++ headers.

- In the user interface, add the folder to your project.

  For more information, see “Add Source and Include Folders” on page 3-30.
- At the command line, use the flag `-I` with the `polyspace-code-prover-nodesktop` command.

For more information, see `-I`.

See Also

More About

- “Errors from Conflicts with Polyspace Header Files” on page 7-56

Language Extensions Supported by Default

Polyspace analysis can interpret a subset of common C/C++ language constructs and extended keywords by default. The default analysis follows these standards:

- C language: C99 Standard (ISO/IEC 9899:1999)

  If you select `Respect C90 standard (-no-language-extensions)`, the analysis follows the C90 Standard.
- C++ language: C++03 Standard (ISO/IEC 14882:2003)

  If you select `C++11 extensions (-cpp11-extension)`, the analysis allows C++11 extensions.

In addition, the default analysis can also interpret language extensions that are supported by many compilers. For other compiler-specific constructs, explicitly specify your compiler.

The analysis can interpret the following constructs, irrespective of your choice of compiler.

- Designated initializers (labeling initialized elements)
- Compound literals (structs or arrays as values)
- Boolean type (`_Bool`)
- Statement expressions (statements and declarations inside expressions)
- `typeof` constructs
- Case ranges
- Empty structures
- Cast to union
- Local labels (`__label__`)
- Hexadecimal floating-point constants
- Extended keywords, operators, and identifiers (`_Pragma`, `__func__`, `__const__`, `__asm__`)

The list is not complete.

In some cases, the analysis supports the construct semantically and fully emulates its run-time behavior. In other cases, the analysis only supports the construct syntactically,

but does not emulate its run-time behavior fully. For instance, the analysis recognizes the construct `asm` as introduction of assembly code, but does not interpret the assembly code encapsulated in the construct. As a result, values modified by the assembly code are considered to have all possible values allowed by their data type.

See Also

Related Examples

- "Specify Target Environment and Compiler Behavior" on page 5-2

More About

- "Supported Keil or IAR Language Extensions" on page 5-10
- "Supported C++ 2011 Language Extensions" on page 5-12

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.

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 7-39.

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.

Supported C++ 2011 Language Extensions

This table lists which C++ 2011 standards Polyspace can analyze. If your code contains non-supported constructions, Polyspace reports a compilation error.

Standard	Description	Supp
C++2011-N2118	Rvalue references	Yes
C++2011-N2439	Rvalue references for *this	Yes
C++2011-N1610	Initialization of class objects by rvalues	Yes
C++2011-N2756	Nonstatic data member initializers	Yes
C++2011-N2242	Variadic templates	Yes
C++2011-N2555	Extending variadic template template parameters	Yes
C++2011-N2672	Initializer lists	Yes
C++2011-N1720	Static assertions	Yes
C++2011-N1984	auto-typed variables	Yes
C++2011-N1737	Multi-declarator auto	Yes
C++2011-N2546	Removal of auto as a storage-class specifier	Yes
C++2011-N2541	New function declarator syntax	Yes
C++2011-N2927	New wording for C++0x lambdas	Yes
C++2011-N2343	Declared type of an expression	Yes
C++2011-N3276	decltype and call expressions	Yes
C++2011-N1757	Right angle brackets	Yes
C++2011-DR226	Default template arguments for function templates	Yes
C++2011-DR339	Solving the SFINAE problem for expressions	Yes
C++2011-N2258	Template aliases	Yes
C++2011-N1987	Extern templates	Yes
C++2011-N2431	Null pointer constant	Yes
C++2011-N2347	Strongly typed enums	Yes
C++2011-N2764	Forward declarations for enums	Yes
C++2011-N2761	Generalized attributes	Yes
C++2011-N2235	Generalized constant expressions	Yes

Standard	Description	Supp
C++2011-N2341	Alignment support	Yes
C++2011-N1986	Delegating constructors	Yes
C++2011-N2540	Inheriting constructors	Yes
C++2011-N2437	Explicit conversion operators	Yes
C++2011-N2249	New character types	Yes
C++2011-N2442	Unicode string literals	Yes
C++2011-N2442	Raw string literals	Yes
C++2011-N2170	Universal character name literals	No
C++2011-N2765	User-defined literals	Yes
C++2011-N2342	Standard Layout Types	No
C++2011-N2346	Defaulted and deleted functions	Yes
C++2011-N1791	Extended friend declarations	No
C++2011-N2253	Extending sizeof	Yes
C++2011-N2535	Inline namespaces	Yes
C++2011-N2544	Unrestricted unions	Yes
C++2011-N2657	Local and unnamed types as template arguments	Yes
C++2011-N2930	Range-based for	Yes
C++2011-N2928	Explicit virtual overrides	Yes
C++2011-N3050	Allowing move constructors to throw [noexcept]	Yes
C++2011-N3053	Defining move special member functions	Yes
C++2011-N2239	Concurrency: Sequence points	No
C++2011-N2427	Concurrency: Atomic operations	No
C++2011-N2748	Concurrency: Strong Compare and Exchange	No
C++2011-N2752	Concurrency: Bidirectional Fences	No
C++2011-N2429	Concurrency: Memory model	No
C++2011-N2664	Concurrency: Data-dependency ordering: atomics and memory model	No
C++2011-N2179	Concurrency: Propagating exceptions	No
C++2011-N2440	Concurrency: Abandoning a process and at_quick_exit	Yes

Standard	Description	Supp
C++2011-N2547	Concurrency: Allow atomics use in signal handlers	No
C++2011-N2659	Concurrency: Thread-local storage	No
C++2011-N2660	Concurrency: Dynamic initialization and destruction with concurrency	No
C++2011-N2340	__func__ predefined identifier	Yes
C++2011-N1653	C99 preprocessor	Yes
C++2011-N1811	long long	Yes
C++2011-N1988	Extended integral types	No

See Also

`C++11 extensions (-cpp11-extension)`

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 them only for the purposes of verification. The following example shows how to remove `far` and `0x` (`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: matlabroot\sys\perl\win32\bin\perl.exe <pathtoscript>\
PostProcessingTemplate.pl
#
##############################################################

$version = 0.1;

$INFILE = STDIN;
$OUTFILE = STDOUT;

while (<$INFILE>)
{

    # Remove far keyword
    s/far//;

    # Remove "@ 0xFE1" address constructs
    s/\@\s0x[A-F0-9]*//g;
```

```
    # Remove "@0xFE1" address constructs
    # s/\@0x[A-F0-9]*//g;

    # Remove "@ ((unsigned)&LATD*8)+2" type constructs
    s/\@\s\(\(unsigned\)\&[A-Z0-9]+\*8\)\+\d//g;

    # 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.

See Also

Polyspace Analysis Options

`Command/script to apply to preprocessed files (-post-preprocessing-command)`

Related Examples

- "Troubleshoot Compilation and Linking Errors" on page 7-7

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 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.
- 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.

For more information, see “Troubleshoot Compilation and Linking Errors” on page 7-7.

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 5.1. 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);
```

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:

Option	Description
`Verify whole application`	The verification stops if the software does not detect a `main`.
`Verify module or library (-main-generator)`	Before verification, Polyspace checks if your code contains a `main` function. If a `main` function exists, the software uses that `main`. Otherwise, the software generates a `main` using the options that you specify: • `Variables to initialize (-main-generator-writes-variables)` • `Initialization functions (-functions-called-before-main)` • `Functions to call (-main-generator-calls)`

Manually Write main Function

During automatic `main` generation, the software makes certain assumptions about the function call sequence or behavior of global variables. For instance, the default automatically generated `main` models the following behavior:

- The functions that you specify using the option `Functions to call (-main-generator-calls)` can be called in arbitrary order.

- In the beginning of each function body, global variables can have the full range of values allowed by their type.

To provide a more accurate model of the call sequence, you can manually write a `main` function for the purposes of verification. You can add this `main` function in a separate file to your project. In some cases, providing an accurate call sequence can reduce the number of orange checks. For example, in the following code, Polyspace assumes that `f` and `g` can be called in any order. Therefore, it produces an orange overflow for the case when `f` is called before `g`. If you know that `f` is called after `g`, you can write a `main` function to model this sequence.

```
static char x;
static int y;

void f(void)
{
    y = 300;
}

void g(void)
{
    x = y;
}
```

Example 1: main Calls One Function Before Another

Suppose you want to verify two functions `func1` and `func2` that have the following prototypes.

```
int func1(void *ptr, int x);
void func2(int x, int y);
```

You know that when both `func1` and `func2` are called, `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, write a `switch` block with a `volatile` condition. For each function, write a `case` branch that calls the function using the `volatile` variable parameters that you created.

   Because each `case` branch is potentially not entered, this condition models the fact that one of `func1` and `func2` might not be called.

For instance, you can write the following `main`:

```
void main()
{
    volatile int random=0;
    volatile void * volatile ptr;
    while(random)
    {
        switch (random)
        {
          case 1:
            random = func1(ptr, random); break;
          default:
            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();

}
```

Verify C++ Classes

In this section...
"Verification of Classes" on page 5-24
"Methods and Class Specifics" on page 5-26

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 alluses 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.

<table>
<tr><th>Code Construct</th><th>Assumption</th></tr>
<tr><td>Global variable</td><td>

Unless explicitly initialized, in each method, global variables can have any value allowed by their type.

For instance, in the following code, Polyspace assumes that `globvar1` can have any value allowed by its type. Therefore, an orange **Division by zero** appears on the division by `globvar1`. However, because `globvar2` is explicitly initialized, the **Division by zero** check on division by `globvar2` is green.

```
extern int fround(float fx);

// global variables
int globvar1;
int globvar2 = 100;

class Location
{
private:
    int x;

public:
    Location(int intx = 0) {
        x = intx;
    };

    void setx(int intx) {
        x = intx;
    };

    void fsetx(float fx) {
        int tx = fround(fx);
        if (tx / globvar1 != 0)
        {
            tx = tx / globvar2;
            setx(tx);
        }
    };
};
```

</td></tr>
</table>

<table>
<tr><th>Code Construct</th><th>Assumption</th></tr>
<tr><td>Classes with undefined constructors</td><td>The members of the classes can be non-initialized.

In the following example, Polyspace assumes that <code>m_loc.x</code> can be non-initialized. Therefore, an orange Non-initialized variable error appears on <code>x</code> in the <code>getMember</code> method. Following the check, Polyspace assumes that the variable can have any value allowed by its type. Therefore, an orange Overflow appears on the addition operation in the <code>show</code> method.

<pre>class OtherClass
{
protected:
    int x;
public:
    OtherClass (int intx);
    int getMember(void) {
        return x;
    };
};

class MyClass
{
    OtherClass m_loc;
public:
    MyClass(int intx) : m_loc(0) {};
    void show(void) {
        int wx, wl;
        wx = m_loc.getMember();
        wl = wx + 2;
    };
};</pre></td></tr>
</table>

Methods and Class Specifics

- "Simple Class" on page 5-27
- "Template Classes" on page 5-28
- "Abstract Classes" on page 5-29
- "Static Classes" on page 5-30
- "Inherited Classes" on page 5-31

- "Simple Inheritance" on page 5-31
- "Multiple Inheritance" on page 5-33
- "Virtual Inheritance" on page 5-34
- "Class Integration" on page 5-34

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

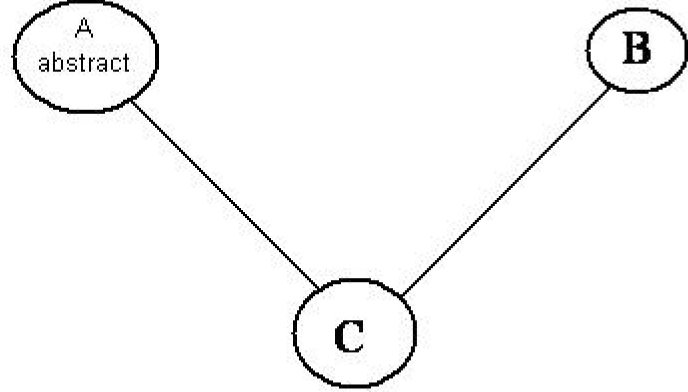
In the real world, an instance of an abstract class cannot be created, so it cannot be analyzed. However, it is easy to establish a verification by removing the pure declarations. For example, this can be accomplished via an abstract class definition change:

`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`:

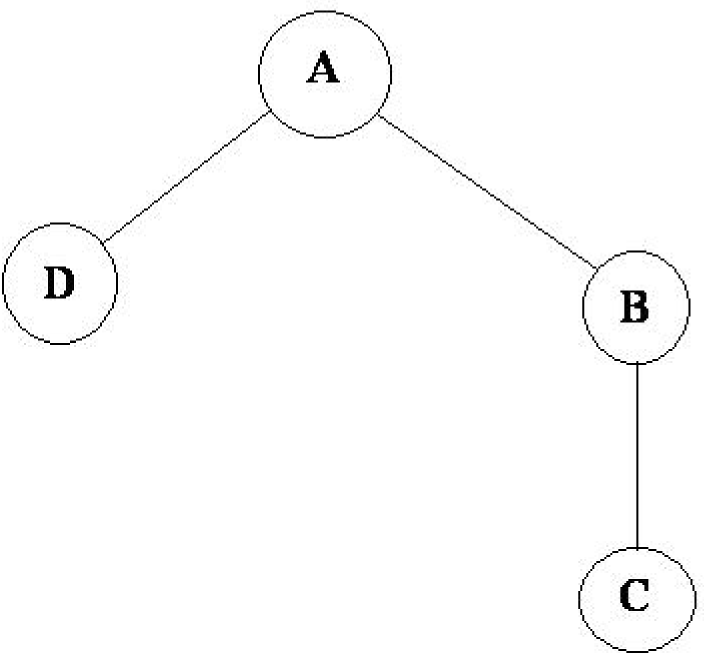
```
class Location
{
protected:
    int x;
    int y;
    Location (int intx, int inty);
public:
    int getx(void) {return x;};
    int gety(void) {return y;};
};
class Point : public Location
{
protected:
    bool visible;
public :
    Point(int intx, int inty) : Location (intx, inty)
    {
    visible = 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:

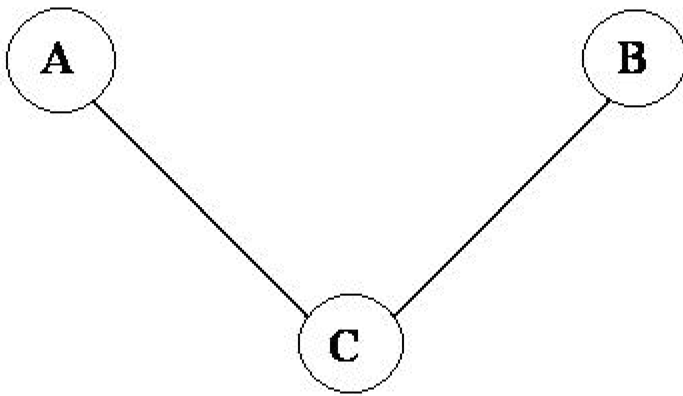
1. You can analyze class A just by providing its code to the software. This corresponds to the previous "Simple Class" section in this chapter.
2. You can analyze class B class by providing its code and the class A declaration. In this case, A code will be stubbed automatically by the software.
3. You 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:

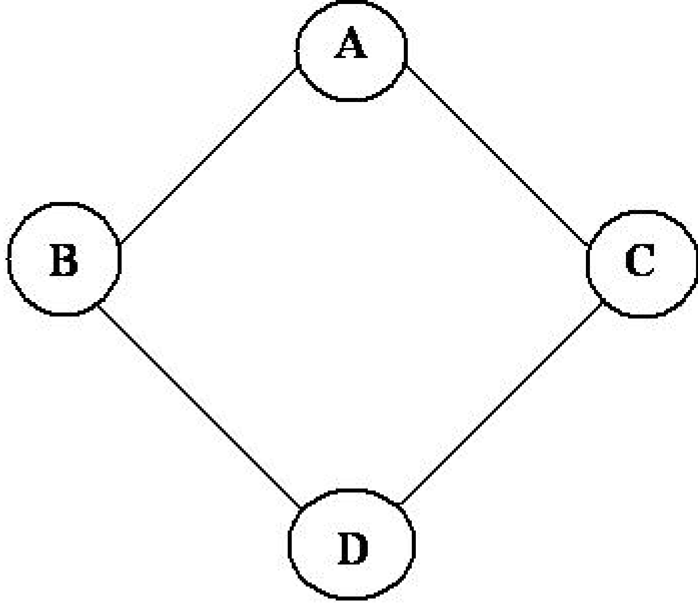
1 You can analyze classes A and B separately just by providing their codes to the software. This corresponds to the previous "Simple Class" section in this chapter.

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.

Specify External Constraints

This example shows how to specify constraints (also known 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.

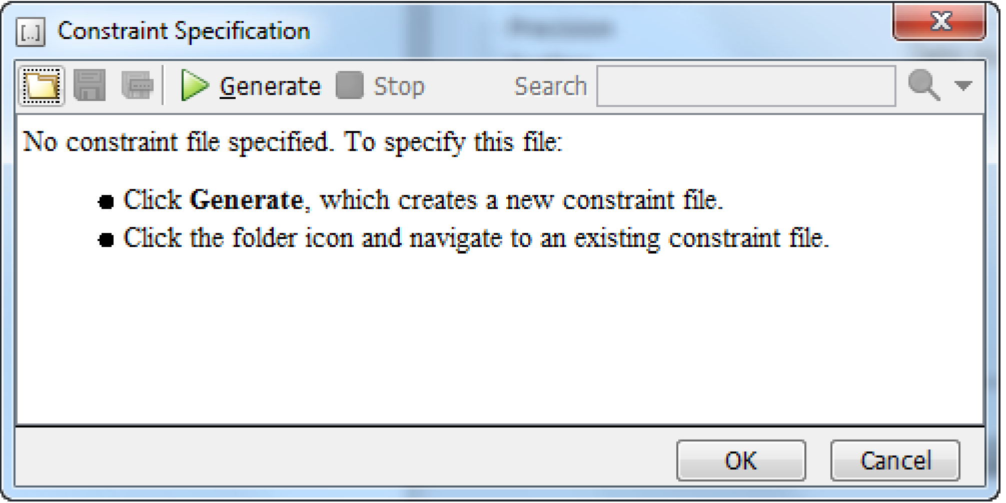
Because of these broad assumptions, Polyspace can consider more execution paths than those paths that occur at run time. If an operation fails along one of the execution paths, Polyspace places an orange check on the operation. If that execution path does not occur at run time, the orange check indicates a false positive.

To reduce the number of such false positives, you can specify additional constraints on global variables, function inputs, and return values of stubbed functions. After you specify your constraints, you can save them 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.

In this section...
“Create Constraint Template” on page 5-35
“Create Constraint Template After Analysis” on page 5-36
“Update Existing Template” on page 5-37
“Specify Constraints in Code” on page 5-38

Create Constraint Template

1. On the **Configuration** pane, select **Inputs & Stubbing**.
2. To the right of **Constraint setup**, click the **Edit** button.



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.

- If you have run analysis once and not changed your code since that analysis, instead of generating a new constraint template, use the folder icon to navigate to the previous results folder. Open the template file `drs_template.xml` from that folder. Save the file in another location, in case you delete the previous results folder.
- Otherwise, to create a new template, click Generate. The software compiles your project and creates a template. The new template is stored in a file *`Module_number_Project_name_`*`drs_template.xml` in your project folder.

4 Specify your constraints and save the template as an XML file. For more information, see "Constraints" on page 5-46.

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.

Create Constraint Template After Analysis

When you create a template based on analysis results, you know which variables you must constrain to avoid false positives.

1. Open your results. Browse orange checks.
2. If the software can trace an orange check to a root cause, a icon becomes available on the **Result Details** pane. Click this icon.

   The root cause is highlighted on the **Orange Sources** pane.
3. If you can address the root cause by using constraints, on the **Orange Sources** pane, in the **Suggestion** column, you see the Add Constraints button. Click this button.
4. On the **Specified Constraints** pane, you see a constraint template. The template contains variables and functions on which you can specify external constraints.

   a. First, save the constraint template as an XML file by using the button.
   b. Specify your constraints.

      For more information, see “Constraints” on page 5-46.
   c. After you specify your constraints, save them using the button in the main toolbar.

   You can use this constraint file for subsequent analyses.

To use the template file for a subsequent analysis, in the project configuration, select **Inputs & Stubbing**. In the **Constraint setup** field, enter the full path to the file.

Update Existing Template

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.

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.

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

- "Constrain Global Variable Range" on page 5-40

More About

- “XML File Format for Constraints” on page 5-56

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 whether write operations on the variable violate the constraint. For the general workflow, see "Specify External Constraints" on page 5-35.

To constrain a global variable range and also check for violation of the constraint:

1. In your project configuration, select **Inputs & Stubbing**. Click the Edit button next to the **Constraint setup** field.
2. In the Constraint Specification window, click Generate.

   Under the **Global Variables** node, you see a list of global variables.
3. For the global variable that you want to constrain:

   - From the drop-down list in the **Global Assert** column, select `YES`.
   - In the **Global Assert Range** column, enter the range in the format *`min..max`*. *`min`* is the minimum value and *`max`* the maximum value for the global variable.
4. To save your specifications, click the button.

   In **Save a Constraint File** window, save your entries as an `xml` file.
5. Run 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.

See Also

Polyspace Analysis Options

`Constraint setup (-data-range-specifications)`

Polyspace Results

`Correctness condition`

Related Examples

- “Constrain Function Inputs” on page 5-42
- “Constrain Stubbed Functions” on page 5-44

More About

- “Constraints” on page 5-46

Constrain Function Inputs

You can specify constraints (also known as data range specifications or DRS) on function inputs. Polyspace checks your function definition for run-time errors in relation to the constrained inputs. For the general workflow, see “Specify External Constraints” on page 5-35.

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) {
      .
      .
}
```

To specify constraints on function inputs:

1. In your project configuration, select **Inputs & Stubbing**. Click the Edit button for **Constraint setup**.
2. In the Constraint Specification window, click Generate.

   Under the **User Defined Functions** node, you see a list of functions that Polyspace does not stub. For information on stubbed functions, see “Constrain Stubbed Functions” on page 5-44.
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 “Constraints” on page 5-46.

   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**.

5. Run verification and open the results. On the **Source** pane, place your cursor on the function inputs.

   The tooltips display the constraints. For example, in the preceding code, the tooltip displays that `val` has values in `1..10`.

See Also

Polyspace Analysis Options

`Constraint setup (-data-range-specifications)`

Related Examples

- “Constrain Global Variable Range” on page 5-40
- “Constrain Stubbed Functions” on page 5-44

More About

- “Constraints” on page 5-46

Constrain Stubbed Functions

Polyspace provides a function stub if you do not define a function or override a function definition using an analysis option.

Polyspace makes certain assumptions about the arguments and return values of stubbed functions. See “Stubbed Functions”. To work around the Polyspace assumptions, you can specify constraints (also known as data range specifications or DRS) on arguments and return values of stubbed functions.

For example, Polyspace assumes that variables returned from undefined functions take full range of values allowed by their type. You can specify that the variable returned by a certain undefined function lies in a specific range.

To specify a constraint, do one of the following:

- Before verification, create a constraint template. Specify this template for verification.

  If you want to specify constraints for all undefined functions, use this approach. For more information, see “Create Constraint Template” on page 5-35.
- Create a constrain template from your verification results. Specify this template for the next verification.

  If you want to constrain only those undefined functions that cause noncritical orange checks, use this approach. For more information, see “Create Constraint Template After Analysis” on page 5-36.

See Also

Polyspace Analysis Options

`Constraint setup (-data-range-specifications)`

Related Examples

- “Constrain Global Variable Range” on page 5-40
- “Constrain Function Inputs” on page 5-42

More About

- “Constraints” on page 5-46

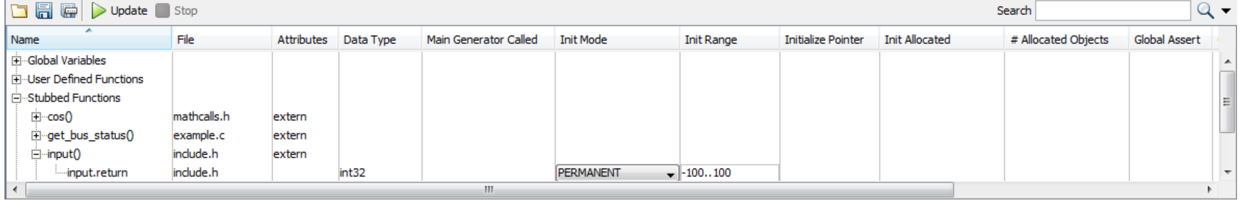
Constraints

Before running a verification, you can define external constraints (also known as data range specifications or DRS) on:

- Global variables.
- User-defined functions.
- Stubbed functions. If you do not define a function or override a function definition using an analysis option, Polyspace provides a function stub. For more information on the analysis option, see `Functions to stub (-functions-to-stub)`.

For information on how to specify constraints in the Polyspace user interface, see "Specify External Constraints" on page 5-35.

The following table lists the constraints that can be specified through this interface. The constraint specification window looks like this:



<table>
<tr><th>Column</th><th>Purpose</th><th>Description</th></tr>
<tr><td>Name</td><td>View the list of variables and functions for which you can specify data ranges.</td><td>This column displays three expandable menu items:
Global Variables – Displays global variables in the project.
User Defined functions – Displays user-defined functions in the project. Expand a function name to see its inputs and return values. The column displays the inputs in separate rows with the syntax <code>function_name.arg1</code>, <code>function_name.arg2</code>, etc.
Stubbed Functions – Displays stubbed functions in the project. Expand a function name to see the inputs and return values. The column displays the inputs in separate rows with the syntax <code>function_name.arg1</code>, <code>function_name.arg2</code>, etc.
Note You cannot specify constraints on all unknown variables in your code. For instance, you cannot constrain pointer arguments of C++ functions.</td></tr>
<tr><td>File</td><td>View the name of the source file containing the variable or function.</td><td></td></tr>
<tr><td>Attributes</td><td>View information about the variable or function.</td><td>Variables can be static or extern. Stubbed functions can be extern.</td></tr>
</table>

Column	Purpose	Description
Data Type	View the data types of variables, function inputs, and function return values.	
Main Generator Called	Specify whether the Polyspace-generated `main` calls a function. If the generated `main` calls a function, Polyspace performs a robustness verification on the function. Otherwise, Polyspace verifies the function in the context of where you call the function in your source code.	The options in this column are: • `MAIN GENERATOR` – The generated `main` calls the function, depending on the `main` generation options that you specify. For more information on these options, see “Code Prover Verification”. • `NO` – The generated `main` does not call the function. • `YES` – The generated `main` calls the function. The options in this column do not apply to stubbed functions. Polyspace provides the body of stubbed functions, therefore calling a stubbed function in the generated `main` for robustness verification is not meaningful.

Column	Purpose	Description

Init Mode	Specify how the software assigns a range to a variable.	The options in this column are:
		• `MAIN GENERATOR` – The generated `main` assigns a range to the variable, depending on the `main` generation options that you specify. For more information on these options, see "Code Prover Verification".
		• `IGNORE` – The generated `main` does not assign a range to the variable, even if you specify a range in the **Init Range** column.
		For pointers, the generated `main` ignores the specifications in the **Initialize Pointer** and **Init Allocated** columns.
		• `INIT` – The generated `main` initializes the variable with the range that you specify.
		This mode is meaningful only for user-defined functions. Use this mode to specify a range on the function inputs. If the function input is not a pointer, you assign the range in the **Init Range** column. If the function input is a pointer, you provide further specifications about the pointer in the **Initialize Pointer** and **Init Allocated** columns.
		• `PERMANENT` – The software permanently assigns the range that you specify to the variable.
		This mode is meaningful only for stubbed functions. For a stubbed function `func` with declaration `int func(int *ptr);`, use this mode to specify a range on the:
		  • Return value of `func`.
		  • Value stored in `*ptr`. This option is available only for C code.
		You specify the ranges in the **Init Range** column.

Column	Purpose	Description

Init Range	Specify a variable range using the syntax *`min_value`* `..` *`max_value`*. The software assigns this range according to the option that you specify in the **Init Mode** column.	Use the keywords `min` and `max` to denote the minimum and maximum values of the variable type. For example, for the type long, `min` and `max` correspond to -2^31 and 2^31-1 respectively.
		
		You can also use hexadecimal values. For example: `0x12..0x100`
		
		For `enum` variables, you cannot specify ranges directly using the enumerator constants. Instead, use the values represented by the constants.
		
		For `enum` variables, you can also use the keywords `enum_min` and `enum_max` to denote the minimum and maximum values that the variable can take. For example, for an `enum` variable of the following type, `enum_min` is 0 and `enum_max` is 5:
		
		`enum week{ sunday, monday=0, tuesday,`
		`     wednesday, thursday, friday, saturday};`

Column	Purpose	Description
Initialize Pointer	Specify whether a pointer can be `NULL` before the first write operation on the pointer.	The option is available only when you specify `INIT` or `PERMANENT` in the **Init Mode** column. The option is meaningful only for pointer arguments of user-defined functions and pointers returned by stubbed functions. The options in this column are: • `May be NULL` – The pointer can have the value `NULL`. If you dereference the pointer before performing a write operation, the software produces an orange **Illegally dereferenced pointer** check. • `Not NULL` – The pointer does not have the value `NULL`. If you dereference the pointer before performing a write operation, the software produces a green **Illegally dereferenced pointer** check. • `NULL` – The pointer has the value `NULL`. If you dereference the pointer before performing a write operation, the software produces a red **Illegally dereferenced pointer** check. To specify that all pointers obtained from external sources can have the value `NULL`, use the option `Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)`. **Note** This column is not applicable to C++ code.

Column	Purpose	Description
Init Allocated	Specify how the software allocates a pointer variable.	The option is meaningful only for: • Pointer arguments of user-defined functions when you specify `INIT` in the **Init Mode** column. • Pointer arguments and return values of stubbed functions. The options in this column are: • `MAIN GENERATOR` – The generated `main` allocates the pointer, depending on the `main` generation options that you specify. For more information on these options, see “Code Prover Verification”. • `NONE` – The software considers that the pointer is not assigned to an object. • `SINGLE` – If the option is applied to a pointer argument of a user-defined function, the software considers that the pointer is assigned to a variable that is certainly initialized. When you read the value that the pointer points to, the software produces a green **Non-initialized variable** check. If the option is applied to a pointer argument or return value of a stubbed function, the software considers that the pointer is assigned to a variable that is potentially initialized. When you read the value that the pointer points to, the software produces an orange **Non-initialized variable** check. You can also use this option to specify an empty string. An empty string consists of a `'\0'` character only and the remaining elements are non-initialized. • `MULTI` – If the option is applied to a pointer argument or return value of an user-defined function, the software considers that the pointer is assigned to an array and that the array elements are certainly initialized. When

Column	Purpose	Description
		you read any array element, the software produces a green **Non-initialized variable** check. If the option is applied to a pointer argument or return value of a stubbed function, the software considers that the pointer is assigned to an array and that the array elements are potentially initialized. When you read any array element, the software produces an orange **Non-initialized variable** check. • `SINGLE_CERTAIN_WRITE` – The option can be applied only to a pointer argument or return value of a stubbed function. The software considers that the pointer is assigned to a variable that is certainly initialized. When you read the value that the pointer points to, the software produces a green **Non-initialized variable** check. • `MULTI_CERTAIN_WRITE` – The option can be applied only to a pointer argument or return value of a stubbed function. The software considers that the pointer is assigned to an array and that the array elements are certainly initialized. When you read any array element, the software produces a green **Non-initialized variable** check **Note** This column is not applicable to C++ code.
# Allocated Objects	Specify the number of objects allocated to a pointer.	The software considers that the pointer points to the first element of an array with the number of elements that you specify. For example, if you specify 3 for a pointer `ptr`, the software produces a green **Illegally dereferenced pointer** check when you access `ptr[0]`, `ptr[1]`, or `ptr[2]`, but produces a red check when you access `ptr[3]`. **Note** This column is not applicable to C++ code.

Column	Purpose	Description
Global Assert	Specify that the software must check whether a global variable exceeds a certain range.	Assign the range in the **Global Assert Range** column. For more information, see “Constrain Global Variable Range” on page 5-40. **Note** If you select `PERMANENT` in the **Init Mode** column, this column is not meaningful because the software permanently assigns a range to the constrained variable.
Global Assert Range	Specify a variable range using the syntax *`min_value`*`..`*`max_value`*.	In the **Global Assert** column, specify whether to enforce the range. For more information, see “Constrain Global Variable Range” on page 5-40. **Note** If you select `PERMANENT` in the **Init Mode** column, this column is not meaningful because the software permanently assigns a range to the constrained variable.
Comment	Enter remarks about the constraints that you specified.	

See Also

Polyspace Analysis Options

`Constraint setup (-data-range-specifications)`

Polyspace Results

`Illegally dereferenced pointer` | `Non-initialized variable`

Related Examples

- “Constrain Global Variable Range” on page 5-40
- “Constrain Function Inputs” on page 5-42

- “Constrain Stubbed Functions” on page 5-44

XML File Format for Constraints

If you run a verification, the software automatically generates a constraint file `drs-template.xml` in your results folder. Edit this XML file to specify your constraints.

Note Instead of editing the constraint XML file directly, use the Polyspace user interface to specify your constraints and save the constraints as an XML file. For more information, see "Specify External Constraints" on page 5-35.

Syntax Description — XML Elements

The DRS 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 5-61.
- **(*****)** — A sub-element of a pointer (i.e. a pointed object) will be taken into account only if `init_pointed` is equal to SINGLE, MULTI, SINGLE_CERTAIN_WRITE or MULTI_CERTAIN_WRITE.
- **(******)** — SINGLE_CERTAIN_WRITE or MULTI_CERTAIN_WRITE are available for parameters and return values of stubbed functions only if they are pointers. If the parameter or return value is a structure and the structure has a pointer field, they are also available for the pointer field.

<file> Element

Field	Syntax
`name`	*`filepath_or_filename`*

Field	Syntax
`comment`	*string*

<scalar> Element

Field	Syntax
`name` **(**)**	*name*
`line` **(*)**	*line*

`base_type` **(*)**	`intx`
	`uintx`
	`floatx`

`Attributes` **(***)**	`volatile`
	`extern`
	`static`
	`const`

`complete_type` **(*)**	*type*

`init_mode`	`MAIN_GENERATOR`
	`IGNORE`
	`INIT`
	`PERMANENT`
	`disabled`
	`unsupported`

`init_modes_allowed` **(*)**	*single value* **(****)**

`init_range`	*range*
	`disabled`
	`unsupported`

`global_ assert`	`YES`
	`NO`
	`disabled`
	`unsupported`

`assert_range`	*range*
	`disabled`
	`unsupported`

`comment`**(*)**	*string*

<pointer> Element

Field	Syntax
Name (**)	*name*
line (*)	*line*
Attributes (***)	volatile extern static const
complete_type (*)	*type*
init_mode	MAIN_GENERATOR IGNORE INIT PERMANENT disabled unsupported
init_modes_allowed (*)	*single value* (****)
initialize_ pointer	May be: NULL Not NULL NULL
number_ allocated	*single value* disabled unsupported
init_pointed (******)	MAIN_GENERATOR NONE SINGLE MULTI SINGLE_CERTAIN_WRITE MULTI_CERTAIN_WRITE disabled

Field	Syntax
comment	*string*

<array> and <struct> Elements

Field	Syntax
Name (**)	*name*
line (*)	*line*
complete_type (*)	*type*
attributes (***)	volatile extern static const
comment	*string*

<function> Element

Field	Syntax
Name (**)	*name*
line (*)	*line*
main_generator_called	MAIN_GENERATOR YES NO disabled
attributes (***)	static extern unused
comment	*string*

Valid Modes and Default Values

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
Global variables	Base type	Unqualified/ static/ const scalar	MAIN_ GENERATOR IGNORE INIT PERMANENT	YES NO			Main generator dependant
		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 dependant
		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			

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
		Pointed extern scalar	INIT	un-supported			INIT min..max
		Pointed other scalars	MAIN_ GENERATO R INIT	un-supported			MAIN_ GENERATO R dependant
		Pointed pointer	MAIN_ GENERATO R INIT/	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	MAIN_ GENERATO R dependant
		Pointed function	un-supported	un-supported			
Function parameter s	Userdef functio n	Scalar parameter s	MAIN_ GENERATO R INIT	un-supported			INIT min..max
		Pointer parameter s	MAIN_ GENERATO R INIT	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May be NULL max MULTI
		Other parameter s	Refer to parameter type				
	Stubbe d functio n	Scalar parameter	disabled	un-supported			

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
		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

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
		Pointer return	PERMANEN T	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI SINGLE_ CERTAIN_ WRITE MULTI_ CERTAIN_ WRITE	PERMANE NT May be NULL max MULTI

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. 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. 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.

Option	Purpose
`Functions to call (-main-generator-calls)`	Specify the functions that the generated `main` must call later.

Control Stubbing Behavior

Use the following options. The options appear under the **Inputs & Stubbing** node.

Option	Purpose
`Functions to stub (-functions-to-stub)`	Specify the functions that Polyspace must stub.

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. 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. The options appear under the **Code Prover Verification** node.

Option	Purpose
`Class (-class-analyzer)`	Specify classes whose methods the generated `main` must call.

Option	Purpose
`Functions to call within the specified classes (-class-analyzer-calls)`	Specify methods that the generated `main` must call.
`Analyze class contents only (-class-only)`	Specify that the generated `main` must call class methods only.
`Skip member initialization check (-no-constructors-init-check)`	Specify that the generated `main` must not check whether each class constructor initializes all class members.

2 Use the following options to call functions that are not class methods. 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.

Verify Multitasking Applications

Your source code can contain functions that are intended to execute concurrently in separate threads (tasks). To verify if your variables are protected against concurrent access, you can specify your tasks and protection mechanisms manually, or allow Code Prover to automatically detect the multitasking model.

Verify Multitasking Code with Automatic Concurrency Detection

If you use POSIX®, VxWorks®, Windows, or µC/OS II functions for multitasking, Code Prover detects your program's multitasking model.

The supported multitasking functions are the following:

Family	Thread Creation	Critical Section Begins	Critical Section Ends
POSIX	`pthread_create`	`pthread_mutex_lock`	`pthread_mutex_unlock`
VxWorks	`taskSpawn`	`semTake`	`semGive`
Windows	`CreateThread`	`EnterCriticalSection`	`LeaveCriticalSection`
µC/OS II	`OSTaskCreate`	`OSMutexPend`	`OSMutexPost`

To enable automatic concurrency detection:

- On the **Configuration** > **Multitasking** pane, select **Enable automatic concurrency detection for Code Prover**.
- (VxWorks only) To activate automatic detection of concurrency primitives for VxWorks, use the `VxWorks` template. For more information on templates, see "Create Project Using Configuration Template" on page 3-25.

For more information and limitations, see `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.

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 `-function-behavior-specifications`.

Configure Multitasking Configuration Manually

This tutorial shows how to set up the multitasking options manually. If the verification does not automatically detect the multitasking primitives used in your code, follow these steps.

1. Before you specify tasks and interrupts to Polyspace, you must code them in a specific format. If your code is already written and does not follow the accepted format, you can make minor adjustments to your code.

   - The tasks must have the following prototype:

     ```
     void func(void)
     ```

   - If you have a `main` function, it must not contain an infinite loop or a run-time error. Polyspace requires that before tasks begin, your `main` function has completed execution.

     For a workaround when your `main` contains an infinite loop, see "Manually Model Tasks if main Contains Infinite Loop" on page 5-72.
   - If want to model a certain scheduling of tasks, you can a create wrapper task to model this sequence.

     For instance, your interrupt occurs only after another task has executed a certain number of times. You can create a wrapper task to model this scheduling and provide only the wrapper task for verification. See "Manually Model Scheduling of Tasks" on page 5-76.
2. On the **Configuration** > **Multitasking** pane, select **Configure multitasking manually**.
3. Specify your entry points, cyclic tasks and interrupts.

   - `Entry points (-entry-points)`: The verification assumes that operations in the entry point function run only once. Use this option to specify initialization tasks that run before other cyclic tasks start execution.
   - `Cyclic tasks (-cyclic-tasks)`: The verification assumes that operations in a cyclic task can run an arbitrary number of times. The operations can be interrupted by operations in entry points, other cyclic tasks and interrupts.
   - `Interrupts (-interrupts)`: The verification assumes that operations in an interrupt can run an arbitrary number of times. The operations cannot be interrupted by operations in entry points, cyclic tasks and other interrupts.

You can also make an entry point or cyclic task nonpreemptable, or an interrupt preemptable. See `-non-preemptable-tasks` and `-preemptable-interrupts`.

4. To protect variables from concurrent access, you must provide specific protection mechanisms in your code.

   - If you want two sections of code to execute without interruption from each other, you can enclose them in the same critical section. Place the two sections of code between calls to the same two functions. Specify these functions using the option `Critical section details (-critical-section-begin -critical-section-end)`.
   - If you want two tasks to execute without interruption from each other, you can specify them to be temporally exclusive. Specify temporal exclusion using the option `Temporally exclusive tasks (-temporal-exclusions-file)`.

   The verification determines if the protection mechanisms are sufficient for protecting shared variables from concurrent access by multiple tasks. When considering possible values of those variables, the verification accounts for the fact that protected sections of your code cannot interrupt each other.

   For a tutorial, see “Manually Protect Shared Variables from Concurrent Access” on page 5-80.

5. Run verification.

   After verification, the **Results List** pane lists potentially unprotected global variables.

   Determine which operations on the variable can occur concurrently. See “Review Global Variable Usage” on page 8-29.

See Also

`Shared protected global variable` | `Shared unprotected global variable`

Related Examples

- “Manually Model Tasks if main Contains Infinite Loop” on page 5-72
- “Manually Model Scheduling of Tasks” on page 5-76

Manually Model Tasks if main Contains Infinite Loop

If you specify your multitasking options manually, this tutorial shows how to model tasks if your `main` function contains an infinite loop. Polyspace requires that before tasks begin, the `main` function has completed execution. If you want your `main` to run concurrently with the tasks instead of completing before them, your `main` function might already contain an infinite loop. If so, for multitasking verification using Polyspace, you must modify your code.

Polyspace Code Prover can detect some multitasking primitives automatically. See `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`. For the high-level workflow on verifying multitasking applications, see “Verify Multitasking Applications” on page 5-69.

For this example, use the following code:

```
void performTask1Cycle(void);
void performTask2Cycle(void);

void main() {
 while(1) {
    performTask1Cycle();
  }
}

void task2() {
 while(1) {
    performTask2Cycle();
  }
}
```

In this example, you learn what happens when you:

1 Specify entry points but retain an infinite loop in `main`.
2 Modify the `main` appropriately so that Polyspace can verify entry point functions.

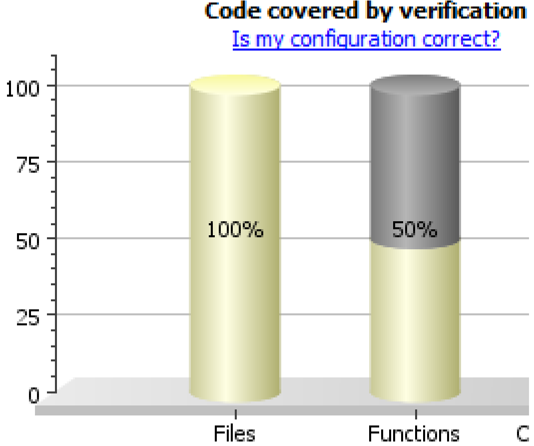
Run Multitasking Verification Without Modifying Code

1 Save the code in a file `multi.c`.
2 Create a Polyspace project and add `multi.c` to it.

3 Specify the following analysis options:

   - Configure multitasking manually: Select this option.
   - Entry points: Enter `task2`.

4 Run verification and open the results. On the **Dashboard** pane, you see that 50% of the functions have been covered.



If you click on the **Functions** column, you see that `task2` is an unreachable procedure.

In addition, if you choose to detect uncalled functions, on the **Results List** pane, you find a gray **Function not reachable** check on `task2`.

Polyspace treats `task2` as not reachable, even though you specified it as an entry point, because the `main` function contains an infinite loop.

Run Multitasking Verification After Modifying Code

1 Replace the following portion of the code

```
void main() {
 while(1) {
    performTask1Cycle();
  }
}
```

with

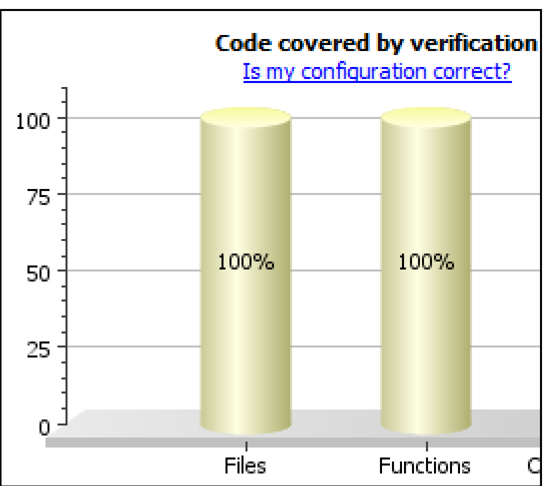
```
#ifdef POLYSPACE
void main() {
}
void task1() {
 while(1) {
    performTask1Cycle();
  }
}

#else
void main() {
 while(1) {
    performTask1Cycle();
  }
}
#endif
```

2 In addition to the options from the previous section, specify the following analysis options:

- Preprocessor definitions: Enter `POLYSPACE`.
- Entry points: Enter `task1` on a separate line.

3 Run verification again. Open the results.

From the **Procedure** column on the **Source** pane **Dashboard**, you find that the verification covered all procedures.



Polyspace verifies both `task1` and `task2` because the `main` function executes to completion.

See Also

`Shared protected global variable` | `Shared unprotected global variable`

Related Examples

- "Manually Model Scheduling of Tasks" on page 5-76
- "Manually Protect Shared Variables from Concurrent Access" on page 5-80
- "Review Global Variable Usage" on page 8-29

More About

- "Verify Multitasking Applications" on page 5-69

Manually Model Scheduling of Tasks

If you specify your multitasking options manually, this tutorial shows how to create a wrapper task for your functions so that they execute in a specific sequence in the task.

Polyspace Code Prover can detect some multitasking primitives automatically. See `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`. For the high-level workflow on verifying multitasking applications, see "Verify Multitasking Applications" on page 5-69.

For this example, save the following code in a file `multi.c`.

```
int var;

void reset(void) {
 var=0;
}

void inc(void) {
  var+=2;
}


void task1(void) {
 volatile int randomValue = 0;
 while(randomValue) {
   inc();
  }
}

void task2(void) {
 volatile int randomValue = 0;
 while(randomValue) {
   reset();
  }
}

void main() {
}
```

In this example, you will learn what happens when you:

1 Specify entry points without modifying your code. The tasks execute in an arbitrary sequence and can interrupt each other any time.

2 Create a new entry point so that the tasks execute in a definite sequence.
3 Modify the new entry point so that each task in the sequence might or might not execute.

Specify Entry Points

1 Create a Polyspace project and add `multi.c` to it.
2 Specify the following analysis options:

   - Configure multitasking manually: Select this option.
   - Entry points: Enter `task1` and `task2` on separate lines.

3 Run verification and open the results.

   An orange **Overflow** error appears on the addition operator in `inc`. The error is not red because it does not occur along all execution paths. The error occurs only if `task1` executes sufficient number of times in succession without interruption from `task2`.

Specify Definite Execution Sequence

Suppose that you want to model that `reset` executes after `inc` has executed five times. This task sequence resets `var` after every five additions and prevents an overflow. To do this:

1 In a separate file `multi_sequence.c`, define a new wrapper function `task` as follows:

```
void task() {
 volatile int randomValue = 0;
 while(randomValue) {
   inc();
   inc();
   inc();
   inc();
   inc();
   reset();
   }
 }
```

2 Add `multi_sequence.c` to the project that you are running verification on.

3 Retain the analysis options from the previous section. However, for Entry points, enter `task` instead of `task1` and `task2`.

4 Run verification and open results.

The orange **Overflow** error does not appear in `inc`. The **Overflow** check is green.

Specify Indefinite Execution Sequence

Suppose, you want to model that `reset` can execute after `inc` has executed zero to five times. This task sequence resets `var` after zero to five additions and also prevents an overflow. To do this:

1 In the file `multi_sequence.c`, modify `task` as follows:

```
void task() {
 volatile int randomValue = 0;
 while(randomValue) {
   if(randomValue)
     inc();
   if(randomValue)
     inc();
   if(randomValue)
     inc();
   if(randomValue)
     inc();
   if(randomValue)
     inc();
   reset();
   }
 }
```

Because `randomValue` is a `volatile` variable, Polyspace considers that the execution can enter or skip any of the five `if` branches.

2 Run verification and open the results.

Again, the **Overflow** check on the addition in `inc` is green.

See Also

`Shared protected global variable` | `Shared unprotected global variable`

Related Examples

- "Manually Model Tasks if main Contains Infinite Loop" on page 5-72
- "Manually Protect Shared Variables from Concurrent Access" on page 5-80
- "Review Global Variable Usage" on page 8-29

Manually Protect Shared Variables from Concurrent Access

If your code uses certain multitasking primitives, Polyspace can interpret these primitives and create a precise multitasking model. Otherwise, you explicitly specify the entry points to your tasks and your protections for shared variables from concurrent access.

In multitasking applications, more than one task can read or write to certain shared variables. When tasks accessing a shared variable execute concurrently, these read and write operations can interrupt each other. Unless you protect the read and write operations, the variable value at a given time is undetermined. To protect variables from concurrent access by multiple tasks, do one or both of the following.

- Specify that certain tasks are temporally exclusive.

  The verification checks if specifying certain tasks as temporally exclusive protects all shared variables from concurrent access. When determining possible values of those shared variables, the verification accounts for the fact that temporally exclusive tasks do not interrupt each other.
- Place read or write access to shared variables inside critical sections. A critical section lies between a call to a lock function and a call to an unlock function. If two sections of code are between the same lock and unlock function, they cannot interrupt each other.

  The verification checks if your placement of lock and unlock functions protects all shared variables from concurrent access. When determining possible values of those shared variables, the verification accounts for the fact that sections of code between the same lock and unlock function do not interrupt each other.

For more information on:

- Automatic detection of multitasking primitives, see `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.
- The high-level workflow for verifying multitasking applications, see “Verify Multitasking Applications” on page 5-69.

Prerequisites

Before starting this tutorial, save the following code in a file `multi.c`. Here, `shared_var` is a variable shared between the tasks,`signal_handler_1` and `signal_handler_2`.

```
#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() {
}
```

View Unprotected Access in Polyspace Results

First, you run verification without specifying protection mechanisms for shared variables. Review the results.

1. Create a Polyspace project and add `multi.c` to it.
2. Specify the following analysis options in your project configuration:

   - Configure multitasking manually: Select this option.
   - `Cyclic tasks (-cyclic-tasks)`: Enter `signal_handler_1` and `signal_handler_2` on separate lines.

3 Run verification and view the following results.

- Orange Shared unprotected global variable.

  The tasks signal_handler_1 and signal_handler_2 access the global variable shared_var. The result indicates that they can access the variable concurrently.

  You can use information in the user interface to determine how the concurrent access takes place. For more information, see "Review Global Variable Usage" on page 8-29.
- Orange Overflow on the line shared_var+=2;.

  The overflow occurs because of the concurrent access. signal_handler_2 assigns INT_MAX or $2^{31}-1$ to shared_var and signal_handler_1 adds 2 to INT_MAX causing the overflow.

  On the **Source** pane, place your cursor on the orange plus sign. You see that the left operand can be $2^{31}-1$

Protect Shared Variables Using Temporally Exclusive Tasks

To protect shared_var from concurrent access, specify that the tasks accessing it are temporally exclusive.

1 In addition to the options from the previous section, for Temporally exclusive tasks, enter signal_handler_1 signal_handler_2.

2 Run verification and view the results.

- Green Shared protected global variable.

  shared_var is protected from concurrent access by signal_handler_1 and signal_handler_2 because signal_handler_1 and signal_handler_2 cannot interrupt each other.
- Green Overflow on the line shared_var+=2;.

  The addition operation in signal_handler_1 cannot take place immediately after signal_handler_2 assigns INT_MAX to shared_var. signal_handler_1 first calls the function reset, and then calls the function inc to perform the addition. Therefore, the addition does not overflow.

On the **Source** pane, place your cursor on the orange plus sign. You see that the left operand can be 0 or 2 only.

Protect Shared Variables Using Critical Sections

To protect `shared_var` from concurrent access, place operations on `shared_var` in a critical section. Use the same critical section in both tasks so that the two sets of operations do not interrupt each other.

1 Modify your code so that operations on `shared_var` are in a critical section. Use the functions `take_semaphore` and `give_semaphore` to begin and end the critical sections. The functions must have the prototype `void` *`func_name`*`(void);`.

The modifications to the code are in bold:

```
#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() {
    }
```

2 In addition to the options from the first section, for Critical section details, enter `take_semaphore` as **Starting routine** and `give_semaphore` as **Ending routine**.
3 Run verification and view the results.

- Green `Shared protected global variable`.

  `shared_var` is protected from concurrent access because the same critical section protects the operations on `shared_var` in `signal_handler_1` and `signal_handler_2`. Therefore, the two sets of operations cannot interrupt each other.
- Green `Overflow` on the line `shared_var+=2;`.

  Because of the protection, the addition operation in `signal_handler_1` cannot take place immediately after `signal_handler_2` assigns `INT_MAX` to `shared_var`. `signal_handler_1` first calls the function `reset`, and then calls the function `inc` to perform the addition. Therefore, the addition does not overflow.

  On the **Source** pane, place your cursor on the orange plus sign. You see that the left operand can be 0 or 2 only.

Tip When you use multiple critical sections, you can encounter issues such as:

- Deadlock: A sequence of calls to lock functions causes two tasks to block each other.
- Double lock: A lock function is called twice in a task without an intermediate call to an unlock function.

Use Polyspace Bug Finder to detect such issues. Then, use Polyspace Code Prover to detect if your placement of lock and unlock functions actually protects the shared variables from concurrent access. For more information, see “Concurrency Defects” (Polyspace Bug Finder).

See Also

Related Examples

- “Manually Model Tasks if main Contains Infinite Loop” on page 5-72
- “Manually Model Scheduling of Tasks” on page 5-76

More About

- `Shared protected global variable`
- `Shared unprotected global variable`

6

Running a Verification

- “Specify Results Folder” on page 6-2
- “Run Local Verification” on page 6-4
- “Run Remote Verification” on page 6-7
- “Phases of Verification” on page 6-10
- “Run File-by-File Local Verification” on page 6-11
- “Run File-by-File Remote Verification” on page 6-14
- “Create Project Automatically at Command Line” on page 6-17
- “Run Local Verification at Command Line” on page 6-19
- “Run Remote Analysis at the Command Line” on page 6-21
- “Modularize Application at Command Line” on page 6-26
- “Scripts for Command-Line Verification” on page 6-29
- “Create Command-Line Script from Project File” on page 6-32
- “Create Project Automatically from MATLAB Command Line” on page 6-35
- “Run Polyspace Analysis by Using MATLAB Scripts” on page 6-37
- “Generate MATLAB Scripts from Polyspace User Interface” on page 6-41
- “Troubleshoot Polyspace Analysis from MATLAB” on page 6-44
- “Storage of Temporary Files” on page 6-46

Specify Results Folder

This example shows how to specify the folder that contains your Polyspace results.

- In the **Project Browser** pane, the folder appears as a node under the **Result** node of your project module.
- This node points to the results folder in your file explorer. The results folder is located in `Project_folder` / `Module_name`. `Project_folder` is the project location that you specified when you created a project.

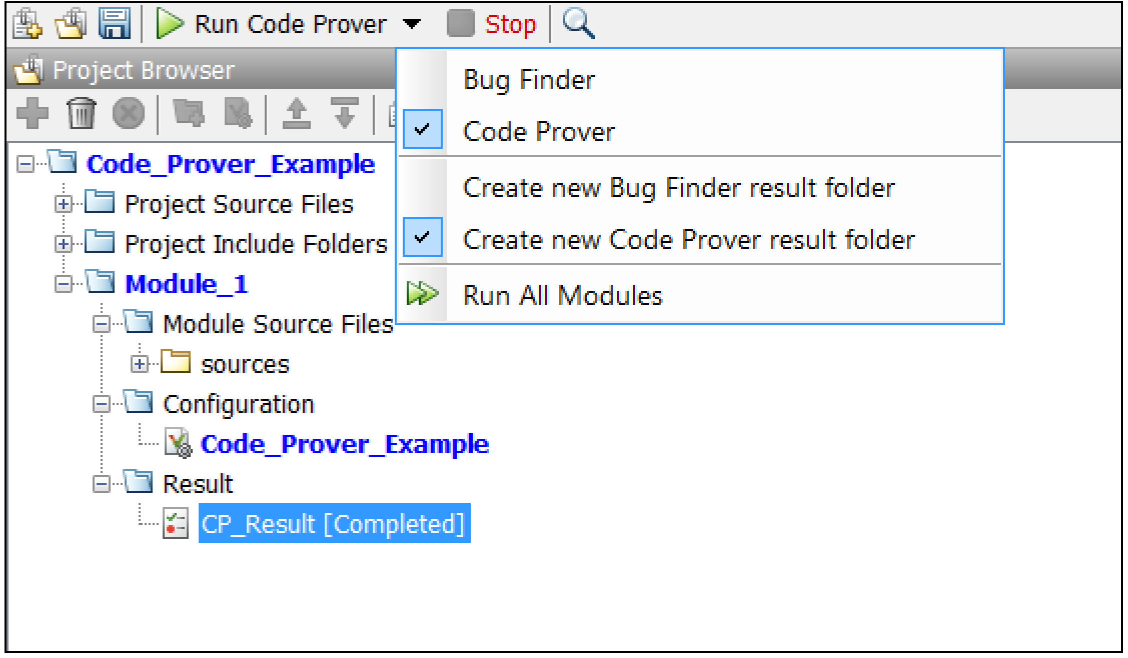
By default, the result folder is named using the convention `CP_Result_n`. *n* is the run number. To rename the folder, right-click the `Result_n` node. To changing the naming convention, select **Tools** > **Preferences**. Create a different naming convention on the **Project and Results Folder** tab.

Create New Result Folder for Each Run

By default, if you rerun an analysis, the previous results are overwritten.

In some cases, you want to store results of a second run in a separate folder from the previous run. For instance, if you change an analysis option and want to see the effect of this change, you want the new results to appear in a separate folder.

To create a new result folder for every run, from the **Run** button drop-down list, select **Create new Code Prover result folder**.



Change Parent Folder of Results Folders

You can also create a parent folder for storing your results. Select **Tools** > **Preferences** and enter the parent folder location on the **Project and Results Folder** tab. If you enter a parent folder location, any new result folder is created under this parent folder.

Run Local Verification

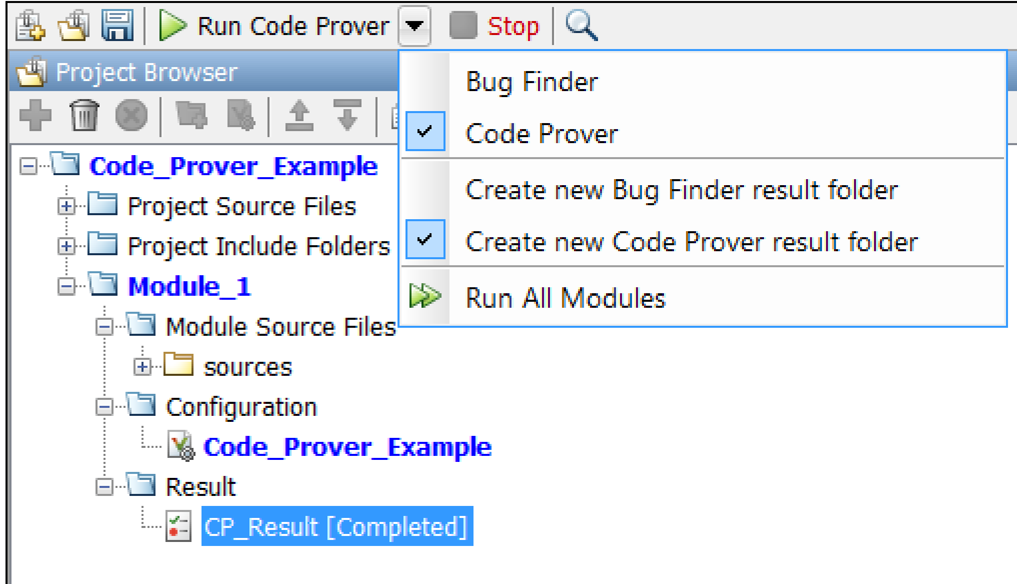
Before running verification on your source files, you must add them to a Polyspace project. For more information, see “Create Project”.

In this section...
“Start Verification” on page 6-4
“Monitor Progress” on page 6-5
“Stop Verification” on page 6-5
“Open Results” on page 6-5

Start Verification

To start a verification on your local desktop:

1 If you do not see the **Run Code Prover** button on the toolbar, select Code Prover as follows.



2 On the toolbar, click **Run Code Prover**.

Tip To run verification on all modules in the project, select **Run All Modules** before starting verification.

Monitor Progress

To monitor the progress of a local verification, use the following panes. If you have closed a pane, to open it again, select **Window > Show/Hide View**.

- **Output Summary** — Displays progress of verification, compile phase messages and errors.
- **Run Log** — This tab displays messages, errors, and statistics for all phases of the verification.

Tip To search for a term in the **Output Summary** or **Run Log**, enter the term on the **Search** pane. Select **Output Summary** or **Run Log** from the drop-down list beside the search box.

If the **Search** pane is not open by default, select **Windows > Show/Hide View > Search**.

At the end of a local verification, the **Dashboard** tab displays statistics, for example, percentage of code checked for run-time errors and check distribution.

Stop Verification

To stop a local verification:

1 On the toolbar, click the **Stop** button.

   A warning dialog box opens asking whether you want to stop the execution.

2 Click **Yes**. The verification stops, and results are incomplete. If you start another verification, the verification starts from the beginning.

Open Results

After verification, the results open automatically on the **Results List** pane. If you are looking at previous results when a verification is over, you can load the new results or retain the previous results on the **Results List** pane.

To open the new results later:

1 On the **Project Browser** pane, navigate to the results set that you want to review.
2 Double-click the results set, for example, **Result_1**.

   The software loads the verification results in the **Results List** pane.

To open results of verification when the corresponding project is not open in the **Project Browser** pane:

1 Select **File** > **Open**.
2 In the Open File dialog box, navigate to the results folder. For example:

   `My_project\Module_1\Result_1`
3 Select the results file, for example, `My_project.pscp`.
4 Click **Open**.

See Also

Related Examples

- “Run File-by-File Local Verification” on page 6-11
- “Run Remote Verification” on page 6-7

More About

- “Phases of Verification” on page 6-10
- “Project and Results Folder Contents” on page 8-131

Run Remote Verification

Run remote verification when:

- You want to shut down your local machine but not interrupt the verification.
- You want to free execution time on your local machine.
- You want to transfer verification to a more powerful computer.

Before you run remote verification, you must do the following:

- Set up a server for this purpose. For more information, see “Set Up Server for Metrics and Remote Analysis”.
- Add your source files to a Polyspace project. For more information, see “Create Project”.

In this section...
“Start Verification” on page 6-7
“Monitor Progress” on page 6-8
“Stop Verification” on page 6-8
“Open Results” on page 6-8

Start Verification

To start a remote verification:

1. Select your project configuration. On the **Configuration** pane, select **Run Settings**. Select **Run Code Prover analysis on a remote cluster**.
2. Optionally, select **Upload results to Polyspace Metrics**.

   After verification, your results are uploaded to the Polyspace Metrics web dashboard.
3. On the toolbar, click the **Run Code Prover** button.

   On the local host computer, the Polyspace Code Prover software performs code compilation and coding rule checking . Then the Parallel Computing Toolbox™ software submits the verification to the MATLAB job scheduler (MJS) on the head node of the MATLAB Distributed Computing Server™ cluster. For more information, see “Phases of Verification” on page 6-10.

Note If you see the message `Verification process failed`, click **OK**. For more information on errors related to remote verification, see “Polyspace Cannot Find the Server” on page 7-23. 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`.

Monitor Progress

You can manage your verification through the Polyspace Job Monitor.

1 Select **Tools** > **Open Job Monitor**.
2 In the Polyspace Job Monitor, right-click your verification.
3 From the context menu, select your management task:
   - **View Log File** — Open the verification log.
   - **Download Results** — Download verification results from remote computer if the verification is complete.

Stop Verification

1 Select **Tools** > **Open Job Monitor**.
2 In the Polyspace Job Monitor, right-click your verification. From the context menu, select **Remove From Queue**.

Open Results

Your results are downloaded automatically after verification. To open them:

1 On the **Project Browser** pane, navigate to the results set.
2 Double-click the results set, for example, **Result_1**.

   The software loads the verification results in the **Results List** pane.

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 13-19.

See Also

Related Examples

- "Run File-by-File Remote Verification" on page 6-14
- "Run Local Verification" on page 6-4

More About

- "Phases of Verification" on page 6-10
- "Project and Results Folder Contents" on page 8-131

Phases of Verification

A verification has three main phases:

1. Checking syntax and semantics (the compile phase). Because Polyspace software is compiler-independent, it helps you to produce code that is portable, maintainable, and compliant with ANSI standards.
2. Generating a `main` function if the Polyspace software does not find a `main` and you have selected the **Verify module or library** option. For more information about generating a main, see `Verify module or library (-main-generator)`.
3. Analyzing the code for run-time errors and generating color-coded results.

Run File-by-File Local Verification

This example shows how to run a local verification on each file independently of other files in the module.

Before running verification on your source files, you must add them to a Polyspace project. For more information, see “Create Project”.

In this section...
“Run Verification” on page 6-11
“Open Results” on page 6-11

Run Verification

1. Select your project configuration. On the **Configuration** pane, specify that each file must be verified independently of other files.

   a. Select the **Code Prover Verification** node.

   b. Select **Verify files independently**.

   c. For **Common source files**, enter files that you want to include in the verification of each file. Enter the full path to a file. Enter one file path per row.

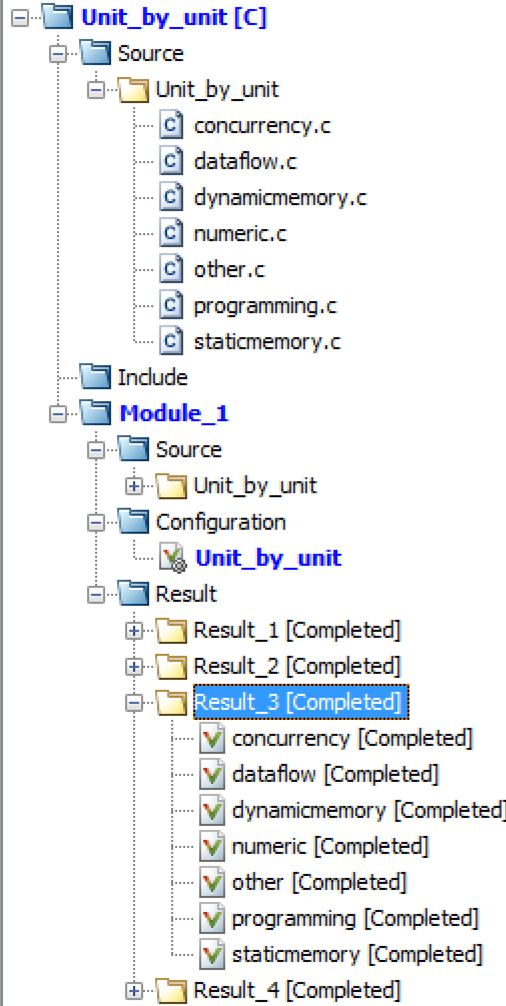
      For example, if multiple files use a function, you must include the file containing the function definition as a common source file. Otherwise, Polyspace stubs the undefined functions leading to more orange checks.

2. On the toolbar, click **Run Code Prover**.

   As you can see on the **Output Summary** pane, after the **Compile** phase, each file is verified independently. After the verification is complete for a file, you can view the results while other files are still being verified.

Open Results

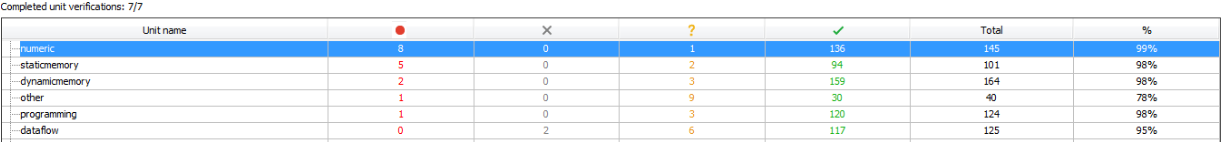
After verification, your results appear in the **Project Browser**. The results are grouped under a root node below the **Result** node of your module. The results for each source file has the same name as the source file.



After a source file is verified, to open the results, double-click the corresponding result file under the **Result** node. Alternatively, after all source files are verified, you can see a summary of results for all files and begin reviewing from files with more severe issues.

1 To open the results after all files are verified, in your project module, click the root node below the **Result** node. For instance, click **Result_3** in the project shown above.

On the **Dashboard** pane, you can see a summary of results for all files.



Completed unit verifications: 7/7

Unit name	●	✕	?	✓	Total	%
numeric	8	0	1	136	145	99%
staticmemory	5	0	2	94	101	98%
dynamicmemory	2	0	3	159	164	98%
other	1	0	9	30	40	78%
programming	1	0	3	120	124	98%
dataflow	0	2	6	117	125	95%
concurrency	0	0	12	50	62	81%

The files in the summary table are sorted by the severity of check colors. For instance, the files are sorted by the number of red checks. The files with the same number of red checks are sorted by the number of gray checks and so on.

2 To load the results for an individual file, double-click the file name on the table.

   After you load the results for an individual file, the **Dashboard** pane shows graphs for the current file. The summary table for all files appears on a separate **Unit by unit results synthesis** tab on this pane. You can use this tab to load results for other files.

3 You can generate a report of the verification results for each file or for all the files together.

   To generate a single report for all the files:

   a Open the results for one file.

   b Select **Reporting** > **Run Report**. Before generating the report, select the option **Generate a single report including all unit results**.

See Also

`Verify files independently (-unit-by-unit)` | `Common source files (-unit-by-unit-common-source)`

Related Examples

- "Run File-by-File Remote Verification" on page 6-14

More About

- "Multiple File Error in File by File Verification" on page 7-79

Run File-by-File Remote Verification

This example shows how to run a remote verification on each file independently of other files in the module.

Before you run remote verification, you must do the following:

- Set up a server for this purpose. For more information, see "Set Up Server for Metrics and Remote Analysis".
- Add your source files to a Polyspace project. For more information, see "Create Project".

In this section...
"Run Verification" on page 6-14
"Open Results" on page 6-15

Run Verification

1. Select your project configuration. On the **Configuration** pane, specify remote verification.

   a. Select the **Run Settings** node.

   b. Select **Run Code Prover analysis on a remote cluster**.

   c. Optionally, select **Upload results to Polyspace Metrics**.

      After verification, your results are uploaded to the Polyspace Metrics web dashboard.

2. On the **Configuration** pane, specify that each file must be verified independently of other files.

   a. Select the **Code Prover Verification** node.

   b. Select **Verify files independently**.

   c. For **Common source files**, enter files that you want to include with verification of each file. Enter the full path to a file. Enter one file path per row.

      For example, if multiple files use a function, you must include the file containing the function definition as a common source file. Otherwise, Polyspace stubs the undefined functions leading to more orange checks.

3 On the toolbar, click the button.

The Parallel Computing Toolbox software submits the verification units as separate jobs to your scheduler. The scheduler is on the head node of the MATLAB Distributed Computing Server cluster.

After the **Compile** phase, you can view the jobs in the Polyspace Job Monitor.

4 Select **Tools** > **Open Job Monitor**.

Your files appear as child nodes under the main verification node. After the verification is complete for a file, you can download and view the results while other files are still being verified. Right-click the row corresponding to the file and select **Download Results**.

Open Results

Your results are automatically downloaded after verification.

To open result for each source file, double-click the corresponding result file under the **Result** node. The result file has the same name as the source file.

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 13-19.

See Also

`Run Bug Finder or Code Prover analysis on a remote cluster (-batch)` | `Verify files independently (-unit-by-unit)` | `Common source files (-unit-by-unit-common-source)`

Related Examples

- "Run File-by-File Local Verification" on page 6-11

More About

- "Multiple File Error in File by File Verification" on page 7-79

Create Project Automatically at Command Line

If you use build automation scripts to build your source code, you can automatically setup a Polyspace project from your scripts. The automatic project setup runs your automation scripts to determine:

- Source files.
- Includes.
- Target & compiler options. For more information on these options, see "Target & Compiler".

Use the `polyspace-configure` command to trace your build automation scripts. You can use the trace information to:

- Create a Polyspace project. You can then open the project in the user interface.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following command to create a Polyspace project `myProject.psprj` from your makefile:

  ```
  polyspace-configure -prog myProject make targetName buildOptions
  ```

  For the list of options allowed with the GNU `make`, see make options.
- Create an options file. You can then use the options file to run analysis on your source code from the command-line.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following commands to create an options file `myOptions` from your makefile:

  ```
  polyspace-configure -no-project -output-options-file myOptions ...
             make targetName buildOptions
  ```

  Use the options file to run analysis:

  ```
  polyspace-code-prover-nodesktop -options-file myOptions
  ```

You can also use advanced options to modify the default behavior of `polyspace-configure`. For more information, see the `-options value` argument for `polyspaceConfigure`.

Note

- In the Polyspace interface, it is possible that the created project will not show implicit defines or includes. The configuration tool does include them. However, they are not visible.
- By default, Polyspace assigns the latest version of the compiler to your project. If you have compilation errors in your project, check the setting for `Compiler (-compiler)`. If it does not apply to you, change it to a more appropriate one.

  For instance, if the compiler setting is `visual12` but you are using Microsoft Visual C++ 2010, change the setting to `visual10`.

  For an example, see "Compilation Error After Creating Project from Visual Studio Build" on page 3-43.
- If your build process requires user interaction, you cannot run the build command from the Polyspace user interface and do an automatic project setup.

See Also

More About

- "Requirements for Project Creation from Build Systems" on page 3-5
- "Compiler Not Supported for Project Creation from Build Systems" on page 3-8
- "Slow Build Process When Polyspace Traces the Build" on page 3-16
- "Check if Polyspace Supports Build Scripts" on page 3-17

Run Local Verification at Command Line

In this section...
"Specify Sources and Analysis Options Directly" on page 6-19
"Specify Sources and Analysis Options in Text File" on page 6-20

Specify Sources and Analysis Options Directly

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

For instance:

- 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-code-prover-nodesktop -sources "file.c" -lang c -target m68k
  ```
- To specify verification precision, use the `-O` option. For instance, to set precision level to 2 for your source file `file.c`, use the command:

  ```
  polyspace-code-prover-nodesktop -sources "file.c" -lang c -O2
  ```

You can specify analysis options multiple times. This flexibility allows you to customize pre-made configurations without having to remove options.

If you specify an option multiple times, only the last setting is used. For example, if your configuration is:

```
-lang c
-prog test_bf_cp
-verif-version 1.0
-author username
-sources-list-file sources.txt
-OS-target no-predefined-OS
-target i386
-compiler generic
-misra-cpp required-rules
-target powerpc
```

Polyspace uses the last target setting, `powerpc`, and ignores the other target specified, `i386`.

For the full list of analysis options, see “Analysis Options”. You can also enter the following at the command line:

```
polyspace-code-prover-nodesktop -help
```

Specify Sources and Analysis Options in Text File

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"
-OS-target no-predefined-OS
-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-nodesktop -options-file listofoptions.txt
```

See Also

More About

- “Scripts for Command-Line Verification” on page 6-29

Run Remote Analysis at the Command Line

Before you run a remote analysis, you must set up a server for this purpose. For more information, see “Set Up Server for Metrics and Remote Analysis”.

In this section...
“Run Remote Analysis” on page 6-21
“Manage Remote Analysis” on page 6-22

Run Remote Analysis

Use the following command to run a remote analysis:

```
matlabroot\polyspace\bin\polyspace-code-prover-nodesktop
-batch -scheduler NodeHost | MJSName@NodeHost [options]
```

where:

- *matlabroot* is your MATLAB installation folder.
- *NodeHost* is the name of the computer that hosts the head node of your MATLAB Distributed Computing Server cluster.
- *MJSName* is the name of the MATLAB Job Scheduler (MJS) on the head node host.
- *options* are the analysis options. These options are the same as that of a local analysis. For more information, see “Run Local Verification at Command Line” on page 6-19.

After compilation, the software submits the analysis job to the cluster and provides you a job ID. Use the `polyspace-jobs-manager` command with the job ID to monitor your analysis and download results after analysis is complete. For more information, see “Manage Remote Analysis” on page 6-22.

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`.

Tip 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 using a `.sh` file.

1. Save your analysis options in a file `listofoptions.txt`. See “Specify Sources and Analysis Options in Text File” on page 6-20.

   To specify your sources, in the options file, instead of `-sources`, use `-sources-list-file`. This option is available only for remote analysis and allows you to specify your sources in a separate text file.
2. Create a file `launcher.bat` in a text editor like Notepad.
3. Enter the following commands in the file.

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

   Where `matlabroot` is your MATLABinstallation folder, and *`localhost`* is the name of the computer that hosts the head node of your MATLAB Distributed Computing Server cluster.
4. Replace the definitions of the following variables in the file:
   - `POLYSPACE_PATH`: Enter the actual location of the `.exe` file.
   - `RESULTS_PATH`: Enter the path to a folder. The files generated during compilation are saved in the folder.
   - `OPTIONS_FILE`: Enter the path to the file `listofoptions.txt`.
5. Double-click `launcher.bat` to run the analysis.

If you run a Polyspace analysis, a Windows `.bat` or Linux `.sh` file is automatically generated for you. The file is in the `.settings` subfolder in your results folder. You can relaunch the analysis using this file.

Manage Remote Analysis

To manage remote analyses, use this command:

```
matlabroot\polyspace\bin\polyspace-jobs-manager action [options]
          [-scheduler schedulerOption]
```

where:

- `matlabroot` is your MATLAB installation folder
- `schedulerOption` is one of the following:

    - Name of the computer that hosts the head node of your MATLAB Distributed Computing Server cluster (`NodeHost`).
    - Name of the MJS 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** > **Server Configuration** > **Job scheduler host name**.
- `action [options]` refer to the possible action commands to manage jobs on the scheduler:

<table>
<tr><th>Action</th><th>Options</th><th>Task</th></tr>
<tr><td><code>listjobs</code></td><td>None</td><td>Generate a list of Polyspace jobs on the scheduler. For each job, the software produces the following information:
<code>ID</code> — Verification or analysis identifier.
<code>AUTHOR</code> — Name of user that submitted job.
<code>APPLICATION</code> — Name of Polyspace product, for example, Polyspace Code Prover or Polyspace Bug Finder.
<code>LOCAL_RESULTS_DIR</code> — Results folder on local computer, specified through the Tools > Preferences > Server Configuration tab.
<code>WORKER</code> — Local computer from which job was submitted.
<code>STATUS</code> — Status of job, for example, <code>running</code> and <code>completed</code>.
<code>DATE</code> — Date on which job was submitted.
<code>LANG</code> — Language of submitted source code.</td></tr>
<tr><td><code>download</code></td><td><code>-job ID -results-folder FolderPath</code></td><td>Download results of analysis with specified ID to folder specified by <code>FolderPath</code>.

If you do not use the <code>-results-folder</code> option, the software downloads the result to the folder you specified when starting analysis, using the <code>-results-dir</code> option.

After downloading results, use the Polyspace user interface to view the results. See “Open Results” on page 6-5.</td></tr>
<tr><td><code>getlog</code></td><td><code>-job ID</code></td><td>Open log for job with specified ID.</td></tr>
<tr><td><code>remove</code></td><td><code>-job ID</code></td><td>Remove job with specified ID.</td></tr>
</table>

Action	Options	Task
`promote`	`-job` *`ID`*	Promote job with specified ID in the queue.
`demote`	`-job` *`ID`*	Demote job with specified ID in the queue.

Modularize Application at Command Line

You can partition large applications into modules. For more information, see "Modularize Project Automatically" on page 3-37.

In this section...
"Basic Options" on page 6-26
"Constrain Module Complexity During Partitioning" on page 6-27
"Result Folder Names" on page 6-27
"Forbid Cycles in Module Dependence Graph" on page 6-28

Basic Options

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

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

This table describes the basic options that you can use.

Option	Description
target_folder	Folder that contains the results of the initial run that processes source files. Default is the folder from which you run `polyspace-modularize`.
`-o=`*output_folder*	Output folder for partitioned application. Default is the folder from which you run `polyspace-modularize`.
`-gui=`*max_n*	The Polyspace verification environment displays the Modularizing choices window with a predefined limit for the maximum number of modules that you can select. Use this option to specify a new limit *max_n*.

Option	Description
`-matlab=`*`max_n`*	If data cache for Modularizing choices window does not exist, create cache *`project_name_max_n`*`.m`. Cache enables faster display of Modularizing choices window. *`project_name`* is the value used by `-prog` option. *`max_n`* is the limit for the maximum number of modules that you can select. No action if cache already exists.
`-modules=`*`n`*	Partition application into *`n`* modules. Identical to clicking the gray region associated with *`n`* in the Modularizing choices window.
`-max-complexity=`*`max_c`*	Partitions application into modules with reference to specified maximum complexity *`max_c`*. The complexity of a function is a number that is related to the size of the function. The complexity of a module is the sum of the complexities of the functions in the module. When partitioning your application, the software minimizes the use of cross-module references to functions and variables, subject to the constraint that the complexity of a module does not exceed *`max_c`*. If you make *`max_c`* sufficiently large, the software generates only one module, which is identical to the original, unpartitioned application.

Constrain Module Complexity During Partitioning

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

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

Result Folder Names

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

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

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

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

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

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

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

Forbid Cycles in Module Dependence Graph

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

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

Scripts for Command-Line Verification

The following sections contain sample scripts that you can use to run a Polyspace verification from the DOS or UNIX command line. For information on the general workflow, see:

- “Run Local Verification at Command Line” on page 6-19
- “Run Remote Analysis at the Command Line” on page 6-21

Simple C Example

```
polyspace-code-prover-nodesktop \
    -prog myCproject \
    -O1 \
    -I /home/user/includes \
    -D SUN4 -D USE_FILES \
```

Apache Example

Here is a script for verifying the code for Apache (after formatting). The source code is in C and the compilation is for an Oracle® Sun™ Microsystems SPARC® processor.

Note The use of O0 to reduce verification time.

```
polyspace-code-prover-nodesktop \ \
    -target sparc \
    -prog Apache \
    -keep-all-files \
    -O0 \
    -D PST \
    -D __GNUC_MINOR__=6 -D SOLARIS2=270 -D USE_EXPAT \
    -D NO_DL_NEEDED \
    -I sources \
    -I /usr/local/pst/include.sparc \
    -I /usr/include \
    -results-dir RESULTS
```

cxref Example

Here is another C launch command. The compilation is for Linux. Note the escape characters, allowing quoted strings to be used as compiler defines.

```
polyspace-code-prover-nodesktop \
    -OS-target linux \
    -prog cxref \
    -O0 \
    -I `pwd` \
    -I sources \
    -I <Polyspace_Install>/include/include.linux \
    -D CXREF_CPP='\"/usr/local/gcc/bin/cpp\"' \
    -D PAGE='\"A4\"' \
    -results-dir RESULTS
```

T31 Example

Another simple C launcher. There are a couple of tasks and compilation is for an m68k.

```
polyspace-code-prover-nodesktop \
    -target m68k \
    -entry-points task_callback_main,task_tcp_main,cdtask_depm_main,task_receiver \
    -to pass1 \
    -prog T31 \
    -O0 \
    -results-dir `pwd`/RESULTS_31 \
```

Dishwasher1 Example

Another C example. This one is for the c-167 and has tasks protected by critical section.

```
polyspace-code-prover-nodesktop \
    -target c-167 \
    -entry-points periodic,pst_main \
    -D PST -D const= -D water= \
    -from scratch \
    -to pass4 \
    -critical-section-begin "critical_enter:cs1" \
    -critical-section-end "critical_exit:cs1" \
    -prog dishwasher1 \
    -I `pwd`/sources \
```

```
    -O0 \
    -results-dir RESULTS
```

Satellite Example

A C example with tasks and critical sections.

```
polyspace-code-prover-nodesktop
    -target c-167 \
    -entry-points ctask0,ctask1,ctask2,ctask3,interrupts \
    -O2 \
    -keep-all-files \
    -from scratch \
    -critical-section-begin "DisableInterrupts:sc1" \
    -critical-section-end "EnableInterrupts:sc1" \
    -ignore-constant-overflows \
    -include `pwd`/sources/options.h \
    -to pass4 \
    -prog satellite \
    -I `pwd`/sources \
    -results-dir RESULTS
```

Create Command-Line Script from Project File

In this section...
“Generate Scripting Files” on page 6-32
“Run an Analysis” on page 6-33

This example shows how to use a project file that you configured in the Polyspace interface to generate the necessary information to run from the command line. If you have already spent time configuring your project in the Polyspace interface, this command is useful to extract your setup work for scripting. For this example, you use the example shipped with Polyspace.

Generate Scripting Files

1. In the Polyspace interface, open the example project by selecting **Help** > **Examples** > **Code_Prover_Example.psprj**.

   This example has been set up and configured with analysis options.
2. Open a command-line terminal and navigate to your `Polyspace_Workspace` folder. By default it is:

   - Linux — `/home/USER/Polyspace_Workspace`
   - Windows — `Users\USER\Documents\Polyspace_Workspace`
   - Mac — `USER/Polyspace_Workspace`
3. Navigate down to the example project:

   ```
   cd Examples/R2017b/Code_Prover_Example
   ```
4. Run the script generation command . (*`matlabroot`* is your installed program folder, for example `C:\Program Files\MATLAB\R2017b`.)

   ```
   matlabroot/polyspace/bin/polyspace-code-prover ...
       -generate-launching-script-for Code_Prover_Example.psprj
   ```

   Polyspace generates the following folder:

   ```
   Code_Prover_Example
   ```

   The folder contains:

- `source_command.txt` — List of source files
- `options_command.txt` — List of the analysis options
- `launchingCommand.sh` (UNIX) or `launchingCommand.bat` (DOS) — Shell script that calls the correct commands

For more details about what files are generated and how to use them, see `-generate-launching-script-for`.

Run an Analysis

After you have completed, “Generate Scripting Files” on page 6-32, you can use the files to run an analysis from the command line. The launching script makes integrating into continuous integration tools such as Jenkins, easier. Here are a few examples of how to use the generated files to run an analysis.

- Run the generated script locally by using the `launchingCommand.bat` file.

  ```
  Code_Prover_Example\launchingCommand.bat
  ```
- Run the generated script and change the results folder.

  ```
  Code_Prover_Example\launchingCommand.bat ...
      -results-dir Results_Code_Prover_Example_RTE_Only
  ```

  The extra `-results-dir` option overrides the results folder specified in the `options_command.txt` file.
- Send the analysis to a remote server and store the results in Polyspace Metrics.

  ```
  Code_Prover_Example\launchingCommand.bat ...
      -add-to-results-repository -batch -scheduler MJS@NoteHost
  ```
- Run the analysis from the command line with the `-options-file` option.

  ```
  matlabroot/polyspace/bin/polyspace-code-prover-nodesktop -options-file ...
      Code_Prover_Example/options_command.txt
  ```

See Also

`-generate-launching-script-for`

Related Examples

- "Create Command-Line Script from Project File" on page 6-32
- "Run Local Verification at Command Line" on page 6-19

External Websites

- How do I use Polyspace with Jenkins?

Create Project Automatically from MATLAB Command Line

If you use build automation scripts to build your source code, you can automatically setup a Polyspace project from your scripts. The automatic project setup runs your automation scripts to determine:

- Source files.
- Includes.
- Target & compiler options. For more information on these options, see “Target & Compiler”.

Use the `polyspaceConfigure` command to trace your build automation scripts. You can use the trace information to:

- Create a Polyspace project. You can then open the project in the user interface.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following command to create a Polyspace project `myProject.psprj` from your makefile:

  ```
  polyspaceConfigure  -prog myProject ...
               make targetName buildOptions
  ```
- Create an options file. You can then use the options file to run analysis on your source code from the command-line.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following commands to create an options file `myOptions` from your makefile:

  ```
  polyspaceConfigure  -no-project -output-options-file myOptions ...
             make targetName buildOptions
  ```

  Use the options file to run analysis:

  ```
  polyspaceCodeProver -options-file myOptions
  ```

You can also use advanced options to modify the default behavior of `polyspaceConfigure`. For more information, see `polyspaceConfigure`.

Note

- In the Polyspace interface, it is possible that the created project will not show implicit defines or includes. The configuration tool does include them. However, they are not visible.
- By default, Polyspace assigns the latest version of the compiler to your project. If you have compilation errors in your project, check the setting for `Compiler (-compiler)`. If it does not apply to you, change it to a more appropriate one.

  For instance, if the compiler setting is `visual12` but you are using Microsoft Visual C ++ 2010, change the setting to `visual10`.

  For an example, see "Compilation Error After Creating Project from Visual Studio Build" on page 3-43.
- If your build process requires user interaction, you cannot run the build command from the Polyspace user interface and do an automatic project setup.

See Also

More About

- "Requirements for Project Creation from Build Systems" on page 3-5
- "Compiler Not Supported for Project Creation from Build Systems" on page 3-8
- "Slow Build Process When Polyspace Traces the Build" on page 3-16

Run Polyspace Analysis by Using MATLAB Scripts

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

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

```
proj = polyspace.Project

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

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

See also `polyspace.Project`.

Specify Multiple Source Files

You can specify a folder containing all your source files. For instance:

```
sourceFolder = fullfile(matlabroot, '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 directly *or in subfolders*. For instance:

```
sourceFolder = fullfile(matlabroot, '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(matlabroot, '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(matlabroot, '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.

```
% 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 = proj.run('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.

To disable MISRA C:2012 rules 8.1 to 8.4:

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

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

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.

```
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 6-44.

Run Analysis on Cluster

You can run an analysis on a cluster 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 “Set Up Server for Metrics and Remote Analysis”.

Specify that the analysis must run on a server. Specify a folder on your desktop where results are downloaded after analysis.

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

Run analysis as usual.

```
proj.run('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 6-41
- “Visualize Polyspace Analysis Results in MATLAB” on page 8-140
- “Troubleshoot Polyspace Analysis from MATLAB” on page 6-44

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.

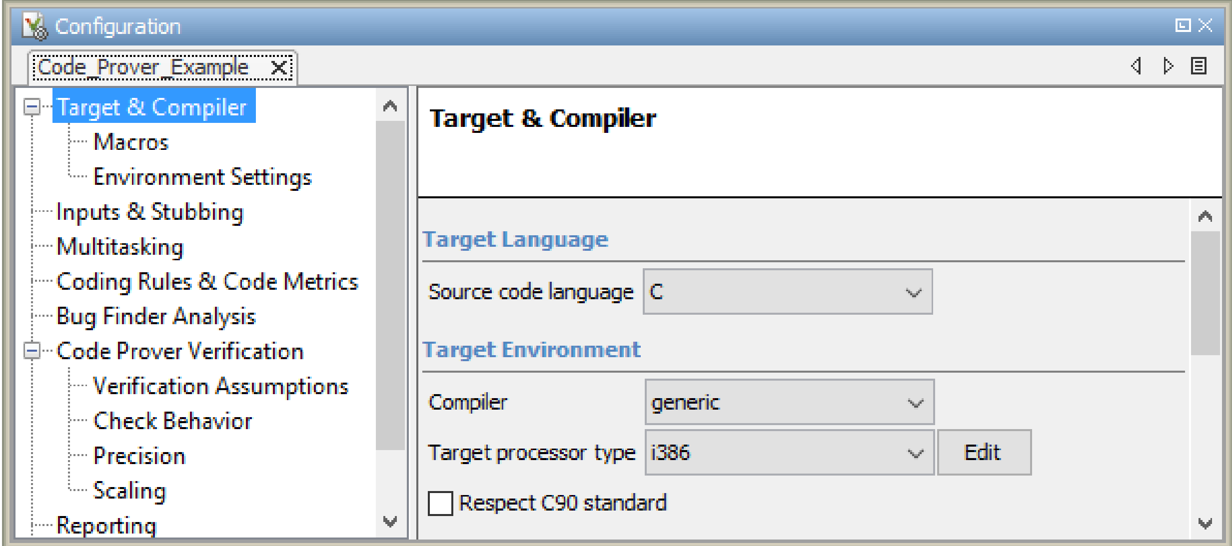
In the user interface, to determine which options to specify, you have tooltips, autocompletion of function names, compilation assistant, context-sensitive help and so on. After you specify the options, you can generate a MATLAB script. For subsequent analyses, you can modify and run the script without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

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 "Create Project".

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.

   For instance, select **Tools** > **Open MATLAB**.

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

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

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 6-37

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.

The following script captures the output from the command window using the `evalc` function.

```
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('proj.run(runMode)');

    % Check log file for compilation errors
    numErrors = 0;
```

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

if numErrors
    errorFile = fopen('error.log','wt+');
    warning('%d compilation error(s). See error.log for details.', numErrors);
    fclose(errorFile);
end

% 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 6-37
- “Troubleshooting in Polyspace Code Prover”

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 7-76
- "Reduce Verification Time" on page 7-12

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

Troubleshooting Verification Problems

- “View Error Information When Analysis Stops” on page 7-3
- “Troubleshoot Compilation and Linking Errors” on page 7-7
- “Reduce Verification Time” on page 7-12
- “Understand Verification Results” on page 7-17
- “Contact Technical Support” on page 7-21
- “Polyspace Cannot Find the Server” on page 7-23
- “Job Manager Cannot Write to Database” on page 7-24
- “Undefined Identifier Error” on page 7-26
- “Unknown Function Prototype Error” on page 7-30
- “Error Related to #error Directive” on page 7-32
- “Large Object Error” on page 7-34
- “Errors Related to Generic Compiler” on page 7-37
- “Errors Related to Keil or IAR Compiler” on page 7-39
- “Errors Related to Diab Compiler” on page 7-40
- “Errors Related to TASKING Compiler” on page 7-43
- “Errors from In-Class Initialization (C++)” on page 7-45
- “Errors from Double Declarations of Standard Template Library Functions (C++)” on page 7-46
- “Errors Related to GNU Compiler” on page 7-47
- “Errors Related to Visual Compilers” on page 7-48
- “Conflicting Declarations in Different Translation Units” on page 7-50
- “Errors from Conflicts with Polyspace Header Files” on page 7-56
- “C++ Standard Template Library Stubbing Errors” on page 7-58
- “Lib C Stubbing Errors” on page 7-59
- “Errors from Assertion or Memory Allocation Functions” on page 7-61
- “Eclipse Java Version Incompatible with Polyspace Plug-in” on page 7-62

- “Reasons for Unchecked Code” on page 7-64
- “Source Files or Functions Not Displayed in Results List” on page 7-69
- “Incorrect Behavior of Standard Library Math Functions” on page 7-73
- “Insufficient Memory During Report Generation” on page 7-75
- “Errors with Temporary Files” on page 7-76
- “Error from Special Characters” on page 7-78
- “Multiple File Error in File by File Verification” on page 7-79
- “Error from Disk Defragmentation and Antivirus Software” on page 7-80
- “License Error –4,0” on page 7-81

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 7-7.

View Error Information in User Interface

1 View the errors on the **Output Summary** tab.

   The messages on this tab appear with the following icons.

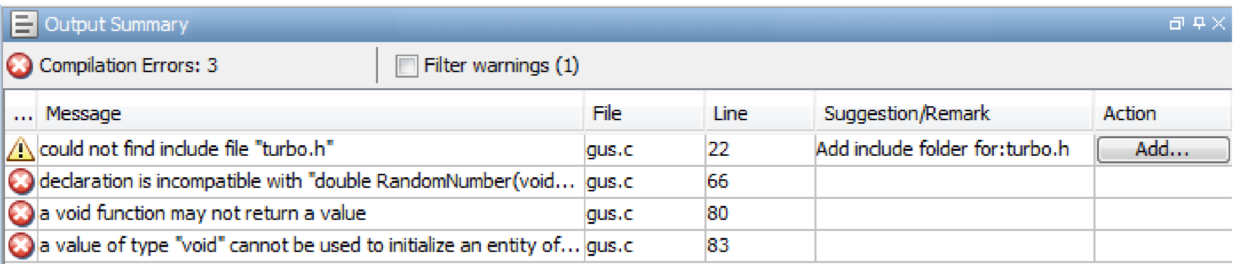
Icon	Meaning
	Error that blocks analysis. For instance, the analysis cannot find a variable declaration or definition and therefore cannot determine the variable type.
	Warning about an issue that does not block analysis by itself, but could be related to a blocking error. For instance, the analysis cannot find an include file that is `#include`-d in your code. The issue does not block the analysis by itself, but if the include file contains the definition of a variable that you use in your source code, you can face an error later.
	Additional information about the analysis.

2 To diagnose and fix each error, you can do the following:

   - To see further details about the error, select the error message. The details appear in a **Detail** window below the **Output Summary** tab.
   - To open the source code at the line containing the error, double-click the message.

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**.

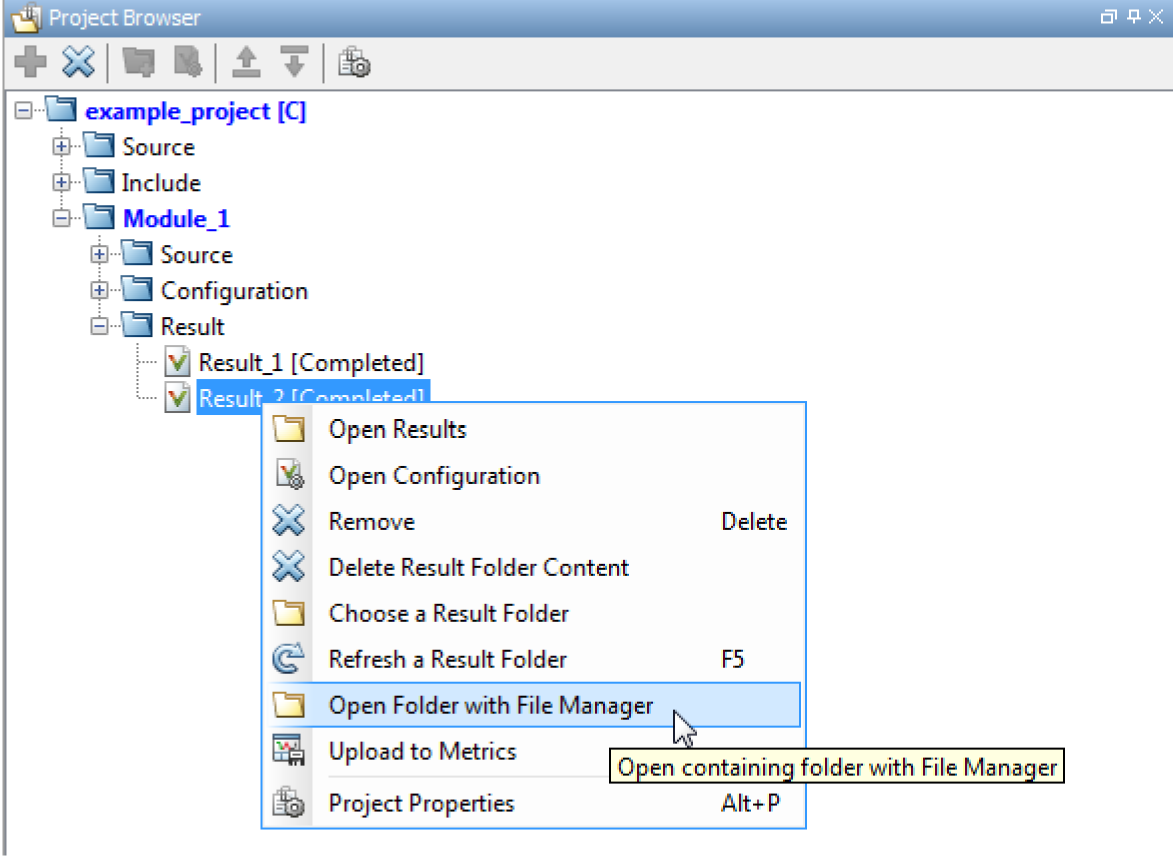
The Compilation Assistant is disabled if you specify the option `Verify files independently (-unit-by-unit)` or `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

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-nodesktop: 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.

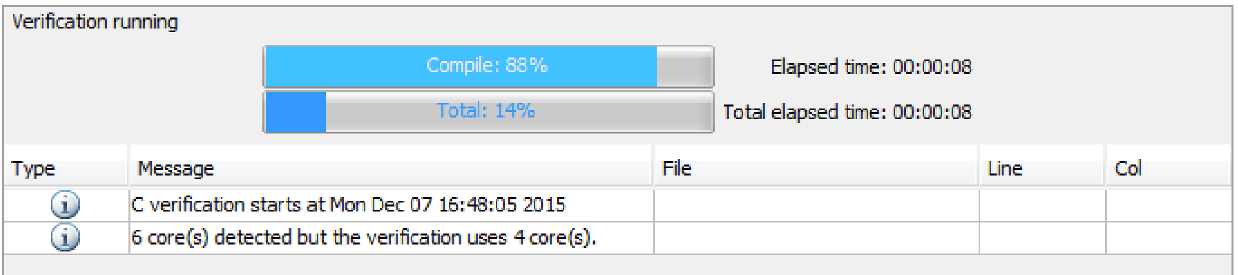
  Add your sources and includes to the project. Change the default analysis options, if required.

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.



Compilation Phase

Type	Message	File	Line	Col
(i)	C verification starts at Thu Dec 17 22:26:17 2015			
(i)	6 core(s) detected but the verification uses 4 core(s).			
(x)	identifier "x" is undefined	my_file.c	1	
(!)	Failed compilation.	my_file.c		
(x)	Verifier has detected compilation error(s) in the code.			
(x)	Exiting because of previous error			

Compilation Failure

Possible Cause: Deviations from ANSI C99 Standard

The Polyspace compiler strictly follows the ANSI C99 Standard (ISO/IEC 9899:1999). 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 7-3.

Otherwise, you can manually adjust your analysis options. To get past compilation issues, use these options.

Option	Purpose
“Target & Compiler” options	Using these predefined options, you can specify your compiler behavior directly and work around known deviations from the standard. Often, setting `Compiler (-compiler)` appropriately is enough to emulate your compiler.
• `Preprocessor definitions (-D)` • `Command/script to apply to preprocessed files (-post-preprocessing-command)`	Using these options, you can sometimes work around unknown deviations from the standard. For instance, you can use these options to replace unrecognized keywords from your preprocessed code with closely matching recognized keywords, or remove them completely. Because you do not change your source code, the options allow you to work around compilation errors while keeping your source code intact.

For specific types of compilation errors, see the *Compilation and Linking* section of “Troubleshooting in Polyspace Code Prover”.

If you cannot solve your compilation error, contact MathWorks Technical Support and provide your compiler name for better support. See “Contact Technical Support” on page 7-21.

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 the *Compilation and Linking* section of “Troubleshooting in Polyspace Code Prover”.

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 *`matlabroot`*`\polyspace\verifier\cxx\polyspace_stubs\`. *`matlabroot`* is the product 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 7-56.

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 7-61.

Reduce Verification Time

In this section...
"Issue" on page 7-12
"Possible Cause: Temporary Folder on Network Drive" on page 7-12
"Possible Cause: Large and Complex Application" on page 7-12
"Possible Cause: Too Many Entry Points for Multitasking Applications" on page 7-15

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.

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 6-46.

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.

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. See “Modularize Project Automatically” on page 3-37.
- If you are running verification in the user interface, you can create multiple modules in your project and copy source files into those modules. See “Modularize Project Manually” on page 3-34.
- 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.

For more information, see “Constrain Stubbed Functions” on page 5-44.

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
- “Review Code Metrics” on page 8-24: How to set limits on code complexity metrics

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)`
Your code contains more than one task and you read a shared variable a large number of times through pointers.	`-lightweight-thread-model`

Possible Cause: Too Many Entry Points for Multitasking Applications

If your code is intended for multitasking and you provide many entry points, 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 `Entry points (-entry-points)`.

For an example of using a wrapper function as an entry point, see “Manually Model Scheduling of Tasks” on page 5-76.

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 “Result and Source Code Colors” on page 8-3.

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.

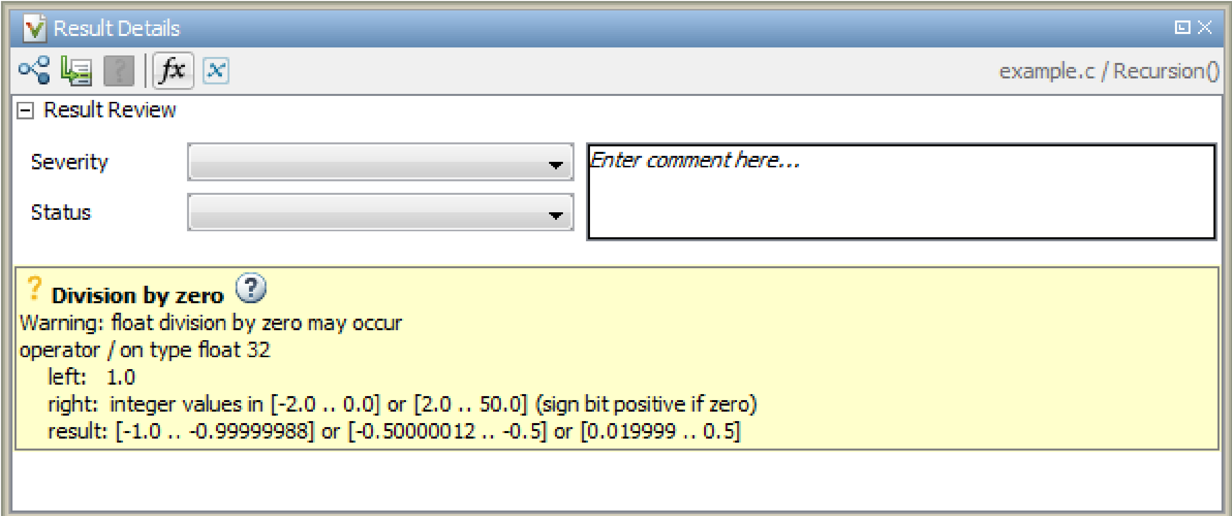
Solution

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 "Types of Run-Time Checks" on page 8-77.

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 *matlabroot*`\polyspace\verifier\code_prover`.

Often, you can change the default assumptions using certain options.

- "Target & 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 & 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 appropriatefor 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" on page 7-21.

Contact Technical Support

To contact MathWorks Technical Support, use this page. You will 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 *matlabroot* with your MATLAB installation folder:
  - UNIX — *matlabroot*/polyspace/bin/polyspace-code-prover-nodesktop -ver
  - Windows — *matlabroot*\polyspace\bin\polyspace-code-prover-nodesktop -ver

Provide Information About the Issue

If you face compilation issues with your project, see “Troubleshooting in Polyspace Code Prover”. 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`*`.txt`. It contains the error message, the options used for the analysis and other relevant information.

- The source files related to the compilation error, 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.

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`*`.txt`. It contains the options used for the analysis and other relevant information.
- The source files related to the result 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 **Preferences** to locate the server. If this information is incorrect, the software cannot locate the server.

Solution

Provide correct server information.

1 Select **Tools** > **Preferences**.
2 Select the **Server Configuration** tab. Provide your server information.

For more information, see “Set Up Server for Metrics and Remote Analysis”.

Job Manager Cannot Write to Database

Message

```
Unable to write data to the job manager database
```

Possible Cause

If the job scheduler cannot send data to the localhost, Polyspace returns this error. The most likely reasons for the MJS being unable to connect to the client computer are:

- Firewalls do not allow traffic from the MJS to the client.
- The MJS cannot resolve the short hostname of the client computer.

Workaround

Add localhost IP to configuration.

1. Select **Tools** > **Preferences**.
2. Select the **Server Configuration** tab.
3. In the **Localhost IP address** field, enter the IP address of your local computer.

   To retrieve your IP address:

   - Windows
     a. Open **Control Panel** > **Network and Sharing Center**.
     b. Select your active network.
     c. 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

- “Set Up Server for Metrics and Remote Analysis”
- “Connection Problems Between the Client and MJS” (Parallel Computing Toolbox)

Undefined Identifier Error

Issue

Polyspace verification fails during the compilation phase with a message about undefined identifiers.

The message indicates that Polyspace cannot find a variable definition. Therefore, it cannot identify the variable type.

Possible Cause: Missing Files

The source code you provided does not contain the variable definition. For instance, the variable is defined in an include file that Polyspace cannot find.

If you `#include`-d the include file in your source code but did not add it to your Polyspace project, you see a previous warning:

```
Warning: could not find include file "my_include.h"
```

Solution

If the variable definition occurs in an include file, add the folder that contains the include file.

- In the user interface, add the folder to your project.

  For more information, see “Add Source and Include Folders” on page 3-30.
- At the command line, use the flag `-I` with the `polyspace-code-prover-nodesktop` 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 5-18.

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 & 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 `Disabled preprocessor definitions (-U)`. 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 7-50.

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, add the folder to your project.

  For more information, see “Add Source and Include Folders” on page 3-30.
- At the command line, use the flag `-I` with the `polyspace-code-prover-nodesktop` 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.

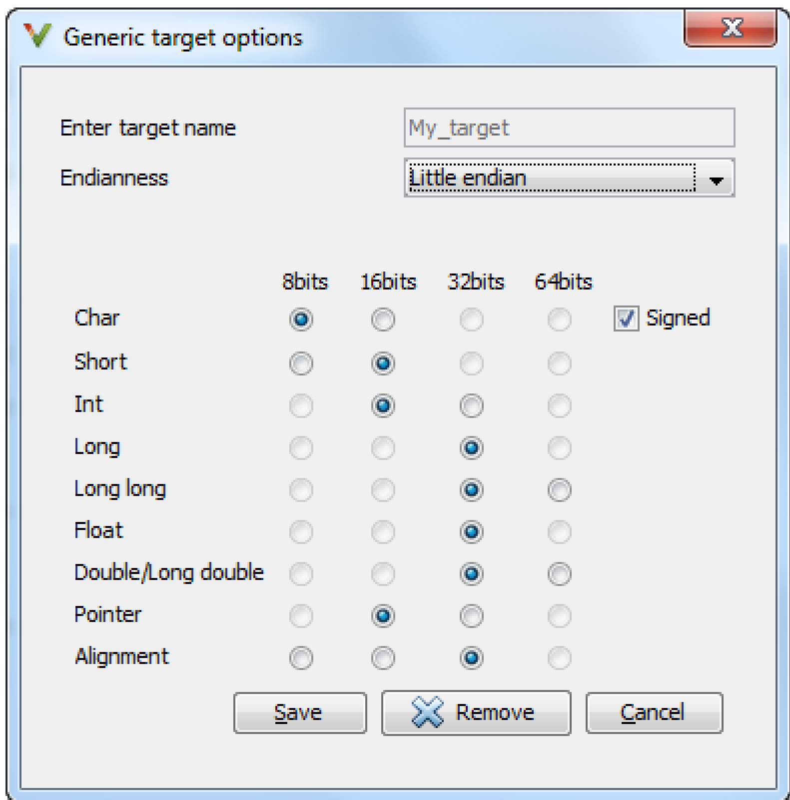
For instance, you get the message, `Limitation: struct or union is too large` in the following example. You specify a pointer size of 16 bits. The maximum object size allocated to a pointer, and therefore the maximum allowed size for an object, can be 2^{16}-1 bytes. However, you declare a structure 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 5-10.

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, 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 TASKING Compiler

If you choose `tasking` for the option `Compiler (-compiler)`, you can encounter this issue.

Issue

During Polyspace analysis, you see an error related to a Special Function Register data type.

Cause

When compiling with the TASKING compiler, you typically use the following compiler flags to specify where Special Function Register (SFR) data types are declared:

- `--cpu=`*`xxx`*: The compiler implicitly `#include`s the file `sfr/reg`*`xxx`*`.sfr` in your source files. Once `#include`-ed, you can use Special Function Registers (SFR-s) declared in that `.sfr` file.
- `--alternative-sfr-file`: The compiler uses an alternative SFR file instead of the regular SFR file. You can use Special Function Registers (SFR-s) declared in that alternative SFR file.

If you specify the TASKING compiler for your Polyspace analysis, the analysis makes the following assumptions about these compiler flags:

- `--cpu=`*`xxx`*: The analysis chooses a specific value of *`xxx`*. If you use a different value with your TASKING compiler, you can encounter an error during Polyspace analysis.

  The *`xxx`* value that the Polyspace analysis uses depends on your choice of `Target processor type (-target)`:

  - `tricore`: `tc1793b`
  - `c166`: `xc167ci`
  - `rh850`: `r7f701603`
  - `arm`: `ARMv7M`
- `--alternative-sfr-file`: The analysis assumes that you do not use an alternative SFR file. If you use one, you can encounter an error.

Solution

Use the command-line option -compiler-parameter in your Polyspace analysis as follows. You use this command-line option to make Polyspace aware of your compiler flags. In the user interface, you can enter the command-line option in the field Other. You can enter the option multiple times.

- --cpu=*xxx*: For your Polyspace analysis, use

  ```
  -compiler-parameter --cpu=xxx
  ```

  Here, *xxx* is the value that you use when compiling with your compiler.
- --alternative-sfr-file: For your Polyspace analysis, use

  ```
  -compiler-parameter --alternative-sfr-file
  ```

  If you still encounter an error because Polyspace is not able to locate your .asfr file, explicitly #include your .asfr file in the preprocessed code using the option Include (-include).

  Typically, the path to the file is *Tasking_C166_INSTALL_DIR*\include\sfr\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 3-5.

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:

<table>
<tr><th>in file Test.h</th><th>in file Test.cpp</th></tr>
<tr><td><pre>class Test
{
public:
static int m_number;
};</pre></td><td><pre>int Test::m_number = 0;</pre></td></tr>
</table>

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, see “Assembly Code”.

Errors Related to Visual Compilers

The following messages appear if the compiler is based on a Visual compiler. For more information, see `Compiler (-compiler)`.

Import Folder

When a Visual application uses `#import` directives, the Visual C++ compiler generates a header file with extension `.tlh` that contains some definitions. To avoid compilation errors during Polyspace analysis, you must specify the folder containing those files.

Original code:

```
#include "stdafx.h"
#include <comdef.h>
#import <MsXml.tlb>
MSXML::_xml_error e ;
MSXML::DOMDocument* doc ;
int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

Error message:

```
"../sources/ImportDir.cpp", line 7: catastrophic error: could not
open source file "./MsXml.tlh"
    #import <MsXml.tlb>
```

The Visual C++ compiler generates these files in its “build-in” folder (usually Debug or Release). In order to provide those files:

- Build your Visual C++ application.
- Specify your build folder for the Polyspace analysis.

pragma Pack

Using a different value with the compile flag (`#pragma pack`) can lead to a linking error message.

Original code:

test1.cpp	type.h	test2.cpp

`#pragma pack(4)`
 	
`#include "type.h"`
	`struct A`
	`{`
	`    char c ;`
	`    int i ;`
	`} ;`
		`#pragma pack(2)`
		
		`#include "type.h"`

Error message:

```
Pre-linking C++ sources ...
"../sources/type.h", line 2: error: declaration of class "A" had
a different meaning during compilation of "test1.cpp"
(class types do not match)
    struct A
           ^
           detected during compilation of secondary translation unit
"test2.cpp"
```

To continue the analysis, use the option `Ignore pragma pack directives (-ignore-pragma-pack)`.

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`
- `Declaration of [...] had a different meaning during compilation of [...] ([...])`

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 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 Bug Finder follows stricter standards for linking to detect violations of system-wide coding rules.

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`.
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 translation units or preprocessed files are stored in a zipped file `ci.zip` in a subfolder `.relaunch` of the results folder.

b Unzip the contents of `ci.zip`.

The preprocessed files have the same name as the source files. For instance, the preprocessed file with `file1.cpp` is named `file1.ci`.

When you open the preprocessed files at the line numbers stated in the error message, you can spot the conflicting declarations.

Possible Cause: Variable Declaration and Definition Mismatch

A variable declaration does not match its definition. For instance:

- The declaration and definition use different data types.
- The variable is declared as signed, but defined as unsigned.
- The declaration and definition uses different type qualifiers.
- The variable is declared as an array, but defined as a non-array variable.
- For an array variable, the declaration and definition use different array sizes.

In this example, the code shows a linking error because of a mismatch in type qualifiers. The declaration in `file1.c` does not use type qualifiers, but the definition in `file2.c` uses the `volatile` qualifier.

file1.c	**file2.c**

`extern int x;`
 	
`void main(void)`
`{/* Variable x used */}`
	`volatile int x;`

In these cases, you can typically spot the difference by looking at the source files. You do not need to see the preprocessed files.

Solution

Make sure that the variable declaration matches its definition.

Possible Cause: Function Declaration and Definition Mismatch

A function declaration does not match its definition. For instance:

- The declaration and definition use different data types for arguments or return values.
- The declaration and definition use a different number of arguments.
- A variable-argument or varargs function is declared in one function, but it is called in another function without a previous declaration.

  In this case, the error message states that the required prototype for the function is missing.

In this example, the code shows a linking error because of a mismatch in the return type. The declaration in `file1.c` has return type `int`, but the definition in `file2.c` has return type `float`.

<table>
<tr><th>file1.c</th><th>file2.c</th></tr>
<tr><td><pre>int input(void);

void main() {
  int val = input();
}</pre></td><td><pre>float input(void) {
  float x = 1.0;
  return x;
}</pre></td></tr>
</table>

In these cases, you can typically find the difference by looking at the source files. You do not need to see the preprocessed files.

Solution

Make sure that the function declaration matches its definition.

Even if your build process allows these errors, you can have unexpected results during run time. If a function declaration and definition with conflicting prototypes exist in your code, when you call the function, the result can be unexpected.

For a variable-argument or varargs function, declare the function before you call it. If you do not want to change your source code, you can work around this linking error.

1 Add the function declaration in a separate file.
2 Only for the purposes of verification, `#include` this file in every source file by using the option `Include (-include)`.

Possible Cause: Macro-dependent Definitions

A variable definition is dependent on a macro being defined earlier. One source file defines the macro while another does not, causing conflicts in variable definitions.

In this example, `file1.cpp` and `file2.cpp` include a header file `field.h`. The header file defines a structure `a_struct` that is dependent on a macro definition. Only one of the two files, `file2.cpp`, defines the macro `DEBUG`. The definition of `a_struct` in the translation unit with `file1.cpp` differs from the definition in the unit with `file2.cpp`.

file1.cpp

```
#include "field.h"

int main()
{
    a_struct s;
    init_a_struct(&s);
    return 0;
}
```

file2.cpp

```
#define DEBUG

#include <string.h>
#include "field.h"

void init_a_struct(a_struct* s)
{
    memset(s, 0, sizeof(*s));
}
```

field.h:

```
struct a_struct {
    int n;
#ifdef DEBUG
    int debug;
#endif
};
```

When you open the preprocessed files `file1.ci` and `file2.ci`, you see the conflicting declarations.

file1.ci	**file2.ci**

`struct a_struct {`
`    int n;`
 	
`};`
	`struct a_struct {`
	`    int n;`
	
	`    int debug;`
	
	`};`

Solution

Avoid macro-dependent definitions. Otherwise, fix the linking errors. Make sure that the macro is either defined or undefined on all paths that contain the variable definition.

Possible Cause: Keyword Redefined as Macro

A keyword is redefined as a macro, but not in all files.

In this example, `bool` is a keyword in `file1.cpp`, but it is redefined as a macro in `file2.cpp`.

file1.cpp

```
#include "bool.h"

int main()
{
    return 0;
}
```

file2.cpp

```
#define false 0
#define true (!false)

#include "bool.h"
```

bool.h:

```
template <class T>
struct a_struct {
    bool flag;
    T t;
    a_struct() {
        flag = true;
    }
};
```

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

```
int main()
{
    return 0;
}
```

file2.cpp

```
#pragma pack(1)

#include "pack.h"
```

pack.h:

```
struct a_struct {
    char ch;
    short sh;
};
```

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 *matlabroot*\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 and Include Folders" on page 3-30.
- At the command line, use the flag `-I` with the `polyspace-code-prover-nodesktop` 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 5-6.

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 and Include Folders” on page 3-30.
   - At the command line, use the flag `-I` with the `polyspace-code-prover-nodesktop` 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 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 7-62
"Cause" on page 7-62
"Solution" on page 7-62

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
matlabroot\sys\java\jre\arch\jre\bin\javaw.exe
```

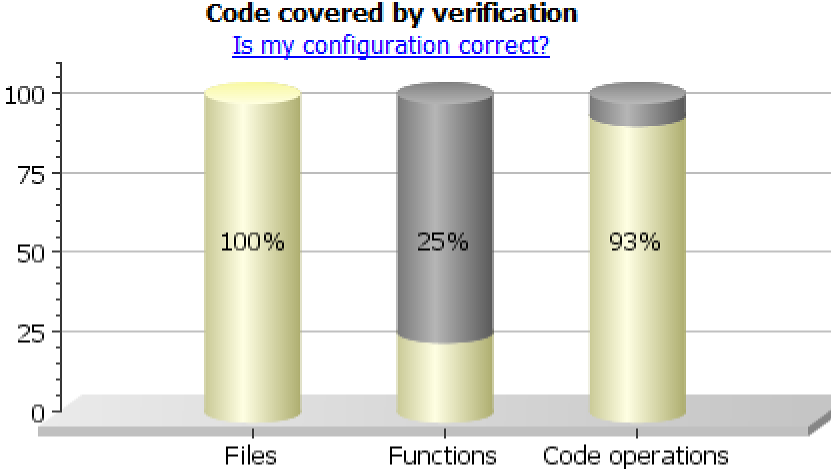
Here, *`matlabroot`* is your product installation folder, for instance, `C:\MATLAB\R2015b\` and *`arch`* is `win32` or `win64` depending on the product platform.

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;
}
```

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:

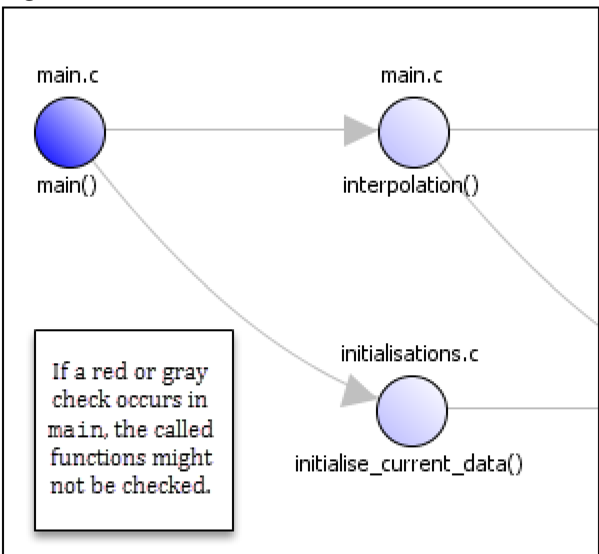
- How the Polyspace compilation works, see “Troubleshoot Compilation and Linking Errors” on page 7-7.
- Specific compilation errors, see the Compilation and Linking section of “Troubleshooting in Polyspace Code Prover”.

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 entry point 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 8-102.

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. See “Review Red Checks” on page 8-10 and “Review Gray Checks” on page 8-15.

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.

Source Files or Functions Not Displayed in Results List

In this section...
“Issue” on page 7-69
“Possible Cause: Files Not Verified” on page 7-69
“Possible Cause: Filters Applied” on page 7-71

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 8-83.

Solution

Polyspace does not verify a source file or function when one of the following situations occur.

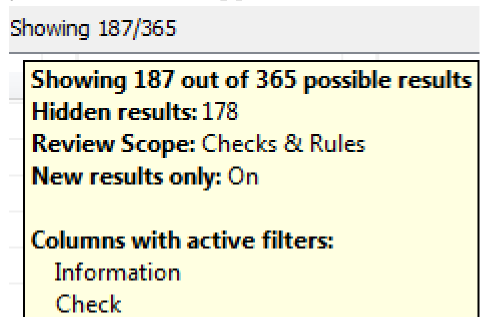
Situation	Fix
The file or function does not contain an operation on which a check is required. For instance, a function contains calls to other functions only. If none of the called functions contains an error that lead to a Non-terminating call error in the calling function, the calling function does not contain a check.	No fix required.
All functions in the source file are not called, are called from unreachable code or are called following red checks. Polyspace does not verify the code that follows a red check and occurs in the same scope as the check. Therefore, it considers that the functions are not called and does not verify the file containing the functions.	If you choose to detect uncalled functions, the verification places a gray check on those functions. The functions and the source file containing the functions then appear on the **Results List** pane. For more information, see `Detect uncalled functions (-uncalled-function-checks)`.
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 `Entry points (-entry-points)`. For a workflow on verifying multitasking code, see “Verify Multitasking Applications” on page 5-69.

Situation	Fix
If your source files do not contain a `main` function, Polyspace generates a `main` function. The generated `main` calls the functions that you specify using certain analysis options. If your analysis options are such that the generated `main` does not call all the functions in a source file, Polyspace does not verify the source file.	See if you have to change the `main` generation options associated with your verification. For more information on the options, see: • `Initialization functions (-functions-called-before-main)` • `Functions to call (-main-generator-calls)` • `Class (-class-analyzer)` • `Functions to call within the specified classes (-class-analyzer-calls).`

Possible Cause: Filters Applied

If you rerun verification on a project module, filters from the last run are applied to the current run. Because of the persistent filters, some of the files can be hidden from display.

To check if some filters are applied, see the **Results List** pane header. The header shows the number of results filtered from the display. If you place your cursor on this number, you can see the applied filters.



For instance, in the image, you can see that the following filters have been applied:

- The **Checks & Rules** filter to suppress code metrics and global variables.

- The New filter to suppress results found in a previous verification.
- Filters on the **Information** and **Check** columns.

Solution

Clear the filters and see if your file or function reappears on the **Results List** pane. For more information, see “Filter and Group Results” on page 8-113.

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 and Include Folders" on page 3-30.

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. For more information on creating projects from build systems, see:

- "Create Project Automatically" on page 3-2
- "Create Project Automatically at Command Line" on page 6-17

- "Create Project Automatically from MATLAB Command Line" on page 6-35

Insufficient Memory During Report Generation

Message

```
....
Exporting views...
Initializing...
Polyspace Report Generator
Generating Report
 .....
    Converting report
Opening log file:  C:\Users\auser\AppData\Local\Temp\java.log.7512
Document conversion failed
.....
Java exception occurred:
java.lang.OutOfMemoryError: Java heap space
```

Possible Cause

During generation of very large reports, the software can sometimes indicate that there is insufficient memory.

Solution

If this error occurs, try increasing the Java heap size. The default heap size in a 64-bit architecture is 1024 MB.

To increase the size:

1. Navigate to `*matlabroot*\polyspace\bin*architecture*`. Where:
   - `*matlab*` 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 `*matlabroot*\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 6-46.
- 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 6-46.
- 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 6-46.
- 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 from Special Characters

Issue

Your file or folder names contain extended ASCII characters, such as accented letters or Kanji characters. You face file access errors during analysis. Error messages you might see include:

- `No source files to analyze`
- `Control character not valid`
- `Cannot create directory` *`Folder_Name`*

Cause

Polyspace does not fully support these characters. If you use extended ASCII in your file or folder names, your Polyspace analysis may fail due to file access errors.

Workaround

Change the unsupported ASCII characters to standard US-ASCII characters.

Multiple File Error in File by File Verification

Issue

When you run a file by file verification, you get the following error message:

```
Verifying unit File1 (2/2)
Error: Unit File1 is defined multiple times
       and has already been treated. Skipped
Warning: Failed compilation of unit: File1
```

For more information on the analysis option for running file by file verification, see `Verify files independently (-unit-by-unit)`.

Possible Cause

You added source files that have the same name.

The files are located in different folders in your file system. Therefore, the conflict does not occur in your file system.

Solution

To perform the file by file verification, work around the file name conflict by:

- Renaming the files.
- Creating a separate module in your Polyspace project. Move the files that cause the file name conflict to that module.

  For more information on creating modules, see "Modularize Project Manually" on page 3-34.

Error from Disk Defragmentation and Antivirus Software

Issue

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 antivirus software is running on your machine.

Solution

Try:

- Stopping the disk defragmentation tool.
- Deactivating the antivirus software. Or, configuring exception rules for the antivirus software to allow Polyspace to run without a failure.

Note Even if the analysis does not fail, the antivirus software can reduce the speed of your analysis. This reduction occurs because the software checks the temporary analysis files. Configure the antivirus software to exclude your temporary folder, for example, `C:\Temp`, from the checking process.

License Error –4,0

Issue

When you try to run Polyspace, you get this error message:

```
License Error -4,0
```

Cause

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.

8

Reviewing Verification Results

- "Result and Source Code Colors" on page 8-3
- "Review Red Checks" on page 8-10
- "Review Gray Checks" on page 8-15
- "Review Orange Checks" on page 8-17
- "Review Code Metrics" on page 8-24
- "Review Global Variable Usage" on page 8-29
- "Add Review Comments to Results" on page 8-32
- "Justify Results Through Code Annotations" on page 8-36
- "Add Review Comments to Code" on page 8-44
- "Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results" on page 8-50
- "Define Custom Annotation Format" on page 8-53
- "Annotation Description Full XML Template" on page 8-62
- "Acronyms for Checks and Code Metrics" on page 8-69
- "Verification Following Red and Orange Checks" on page 8-72
- "Types of Run-Time Checks" on page 8-77
- "HIS Code Complexity Metrics" on page 8-81
- "Result Views in Polyspace User Interface" on page 8-83
- "Filter and Group Results" on page 8-113
- "Prioritize Check Review" on page 8-119
- "Software Quality Objectives" on page 8-122
- "Project and Results Folder Contents" on page 8-131
- "Generate Report" on page 8-133
- "Export Polyspace Analysis Results" on page 8-136
- "Visualize Polyspace Analysis Results in MATLAB" on page 8-140
- "Customize Existing Report Template" on page 8-144

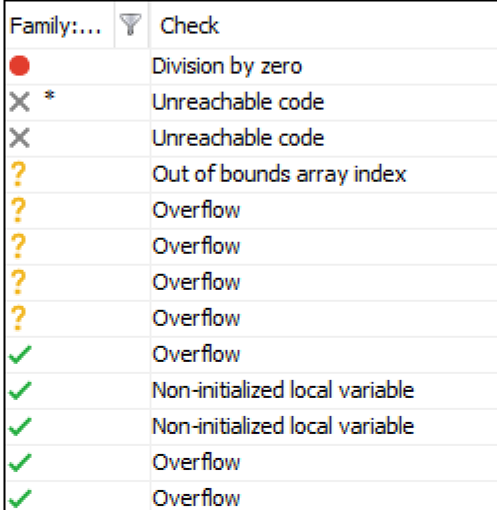
- "Sample Report Template Customizations" on page 8-150
- "Set Character Encoding Preferences" on page 8-154

Result and Source Code Colors

This topic explains the various colors used in displaying the results of a Polyspace Code Prover analysis.

Result Colors

Polyspace displays the different verification results with specific icons and colors on the **Results List** and **Result Details** pane.



Run-Time Checks

Polyspace Code Prover checks each operation in your code for particular run-time errors. The software assigns a color to the operation based on whether it proved the existence or absence of a run-time error on all or some execution paths.

<table>
<tr><th>Check Color</th><th>Purpose</th><th>Example</th><th>Icon</th></tr>
<tr><td>Red</td><td>Highlights operations that are proven to cause a particular error on all execution paths*.

Polyspace Code Prover verification determines errors with reference to the language standard. Though some of the errors can be acceptable for a particular compilation environment, they violate the language standard. To allow some of the environment-dependent behavior, use appropriate analysis options. For more information, see “Verification Assumptions” and “Check Behavior”.</td><td>Red Overflow on:

<code>z = x+y;</code>

The operation + overflows for every value of <code>x</code> and <code>y</code> that the verification considers at that point.</td><td></td></tr>
<tr><td>Gray</td><td>Highlights unreachable code.</td><td>Gray Unreachable code check:

<code>if(x>0)</code>
<code>{}</code>
<code>else</code>
<code>{}</code>

The <code>else</code> branch is unreachable for all values of <code>x</code> that the verification considers at that point.</td><td></td></tr>
</table>

Check Color	Purpose	Example	Icon
Orange	Highlights operations that can cause an error on certain execution paths. For more information, see “Sources of Orange Checks” on page 10-2.	Orange **Overflow** on: `z = x+y;` The analysis could not prove whether the operation + overflows. The most common reason is that the operation overflows only for some values of `x` and `y` that the verification considers at that point. You can use the tooltips on the variables `x` and `y` in the operation to see the range of values that the verification considers.	?
Green	Highlights operations that are proven to not cause a particular error on all execution paths*.	Green **Overflow** on: `z = x+y;` The operation + does not overflow for all values of `x` and `y` that the verification considers at that point.	✓

* For most checks, the software terminates an execution path following the first run-time error on the path. Therefore, if it proves a definite error (red) or absence of error (green) on an operation, the proof is valid only for the execution paths that have not yet been terminated at that point in the code. See “Verification Following Red and Orange Checks” on page 8-72.

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.
Code metrics	Indicates code complexity metrics.	for metrics that do not exceed a limit you specified and ! for metrics that exceed a limit.
Global variables	Indicates global variable declaration.	? for shared potentially unprotected variables and X for non-shared unused variables

Source Code Colors

Polyspace uses the following color scheme for displaying code on the **Source** pane.

- *Lines with checks*:

  For every check on the **Results List** pane, Polyspace assigns the check color to the corresponding section of code.

  - For lines containing macros, if the macro is collapsed, then Polyspace colors the entire line with the color of the most severe check on the line. The severity decreases in this order: red, gray, orange, green.

    This unreachable `for` loop contains a macro `MAX_SIZE`. The entire line is colored gray.

    ```
    for (i = 0; i < MAX_SIZE; i++) {
    ```

    If there is no check in a line containing a macro, Polyspace underlines the line in black when the macro is collapsed.
  - For all other lines, Polyspace colors only the keyword or identifier associated with the check.

    This assignment has three checks: `i` and `used_global` are initialized but the array `tab` can be accessed outside its bounds. The `[` operator is colored orange to indicate the issue.

    ```
    tab[i] = used_global;
    ```

- *Lines with coding rule violations*:

  For every coding rule violation on the **Results List** pane, Polyspace assigns to the corresponding keyword or identifier:

  - A ▿ 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 to indicate that the function body contains a definite run-time error. The tooltip shows the line in the function 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*:

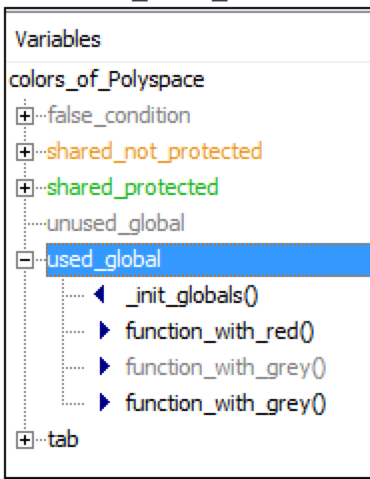
  Polyspace assigns a lighter shade of gray to code deactivated due to conditional compilation. Such code occurs, for instance, in `#ifdef` statements where the macro for a branch is not defined. This code does not affect the verification.

```
#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 written 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 8-104.

Review Red Checks

During verification, Polyspace Code Prover checks each operation in your code for certain run-time errors. After verification, the software displays the checks on the **Results List** pane.

A red check indicates that the operation fails the check on all execution paths. For instance, a red **Division by Zero** check on a division operation indicates that a division by zero occurs every time the operation takes place. Therefore, you must fix the code containing a red check.

For details of the steps for each check type, see “Types of Run-Time Checks” on page 8-77.

Red checks in a block of code stop verification of the remaining code in the block. At the cost of missing certain run-time errors, you can disable some checks and continue verification. For more information on the options to disable checks, see Check Behavior (C).

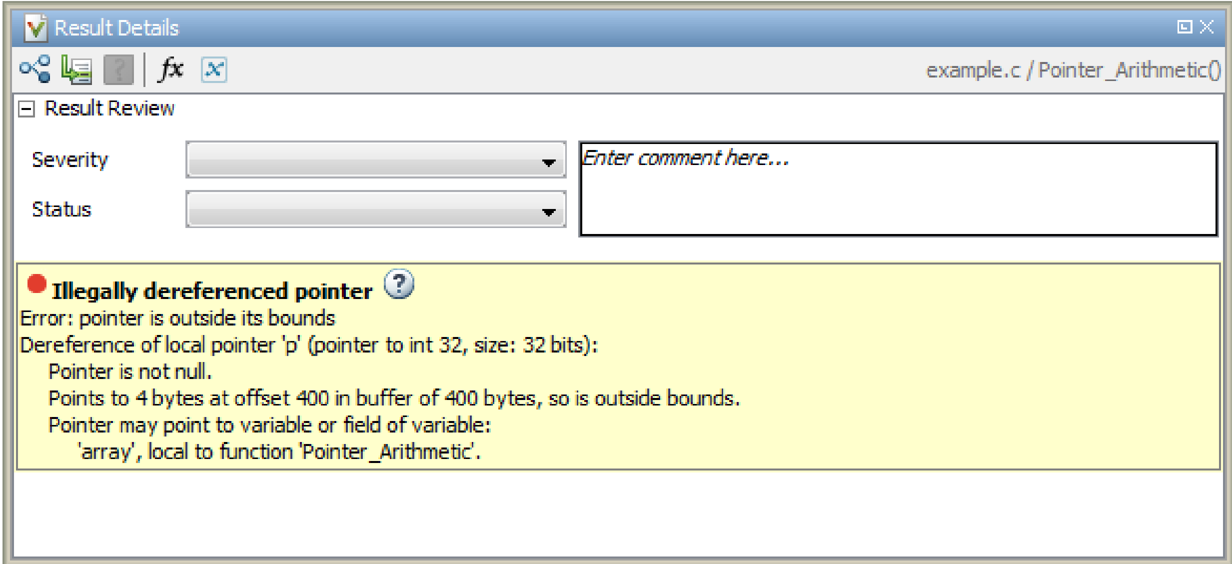
Step 1: Interpret Check Information

1 Select a check on the **Results List** pane.

   - On the **Result Details** pane, view further information about the check.
   - On the **Source** pane, the operation containing the check is highlighted.

     If you place your cursor on the operation, the tooltip provides further information about the check.

   Sometimes, this information is sufficient to understand the root cause of the check. If you can determine a fix for your code from this information, you do not have to proceed further with this procedure.



For details on the information available for each check type, see "Types of Run-Time Checks" on page 8-77.

2 Sometimes, the **Result Details** pane lists the sequence of instructions that led to the check. If you see this sequence, select each instruction to trace back to the root cause of the check in your source code.

Step 2: Determine Root Cause of Check

If you cannot determine the root cause based on the check information, using navigation shortcuts in the user interface, navigate to the root cause.

For details on the navigation process for each type of check, see "Types of Run-Time Checks" on page 8-77. The high-level workflow is:

1 Using the tooltips on variables or operations, identify the variable `var` that causes the check. For instance, for an **Out of bound array index** error, `var` can be the array index.
2 Trace the data flow for `var`.

   a Browse through the previous instances of `var`. On the **Source** pane, place your cursor on each instance of `var` to see its values.

Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. i Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. ii On the **Search** pane, select the previous instances. • Browse the source code. i Double-click the variable on the **Source** pane. All instances of the variable are highlighted. ii 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.	i Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. ii On this pane, select the previous instances of the variable. Write operations on the variable are indicated with ◀ and read operations with ▶.

Variable	How to Find Previous Instances of Variable
Function return value `ret=func();`	i Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration. ii In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 9-104.

b Find the instance where `var` acquires the value that can cause the run-time error.

3 If `var` obtains values from another variable, trace the data flow for the second variable.

Continue this process until you identify the root cause of the check.

For details on the navigation process for each check type, see “Types of Run-Time Checks” on page 8-77.

Step 3: Look for Common Causes of Check

Try to methodically determine the root cause for each check. Polyspace Code Prover lists the vulnerabilities in your code. Unless you methodically address these code vulnerabilities, you can miss possible run-time errors.

If you are aware of typical coding errors that cause a red or orange check, root cause investigation is often easier. You can browse through your source code to look for those errors.

For details on the common coding errors for each check type, see “Types of Run-Time Checks” on page 8-77.

See Also

Related Examples

- “Review Gray Checks” on page 8-15
- “Review Orange Checks” on page 8-17
- “Review Coding Rule Violations” on page 12-13
- “Review Code Metrics” on page 8-24
- “Review Global Variable Usage” on page 8-29

More About

- “Results Management”
- “Result Views in Polyspace User Interface” on page 8-83

Review Gray Checks

Gray checks indicate code that cannot be reached during run-time. Polyspace Code Prover runs three checks for unreachable code:

- `Unreachable code`
- `Function not called`

  This check is not turned on by default.
- `Function not reachable`

  This check is not turned on by default.

If the gray check indicates defensive code, ignore the check. For instance, you can have error handling tests in your code. If the errors do not occur, the test blocks appear gray. However, you might want to retain the error handling test.

In some cases, unreachable code results from coding errors. Therefore, you must review the gray checks. Also, if you do not want to retain unnecessary code, review and fix gray checks.

Note Following a red check, Polyspace does not verify the remaining code in the same scope as the check. However, this code does not appear gray on the **Source** pane.

Review and fix the red checks so that Polyspace can verify the remaining code. For more information, see “Review Red Checks” on page 8-10.

---

1. After verification, see the **Code covered by verification** graph on the **Dashboard** pane. The graph displays the percentage of files, functions and code operations checked for run-time errors.
2. If the percentage of functions covered is less than 100, investigate why there are unreachable functions.

   - Select the column graph to see a list of unreachable functions.
   - If you want to justify uncalled and unreachable functions, rerun verification using appropriate values for the option **Detect uncalled functions**. The uncalled and unreachable functions appear as gray checks on the **Results List** pane.

For more information, see `Detect uncalled functions (-uncalled-function-checks)`.

3. Investigate the checks further. For more information, see:

   - "Review and Fix Unreachable Code Checks" on page 9-93
   - "Review and Fix Function Not Called Checks" on page 9-16
   - "Review and Fix Function Not Reachable Checks" on page 9-19

4. If you determine that the check represents defensive code, ignore the check. Add a comment and justification in your result or code explaining the rationale.

   For more information, see:

   - "Add Review Comments to Results" on page 8-32
   - "Add Review Comments to Code" on page 8-44

See Also

Related Examples

- "Review Red Checks" on page 8-10
- "Review Orange Checks" on page 8-17
- "Review Coding Rule Violations" on page 12-13
- "Review Code Metrics" on page 8-24
- "Review Global Variable Usage" on page 8-29

More About

- "Results Management"
- "Result Views in Polyspace User Interface" on page 8-83

Review Orange Checks

During verification, Polyspace Code Prover checks each operation in your code for certain run-time errors. After verification, the software displays the checks on the **Results List** pane.

An orange check indicates that the operation fails the check only on certain execution paths. For more information, see “Sources of Orange Checks” on page 10-2.

Investigate whether the execution paths can occur during run time. If you determine that the execution paths can occur, you must fix the code containing the check.

For details of the steps for each check type, see “Types of Run-Time Checks” on page 8-77.

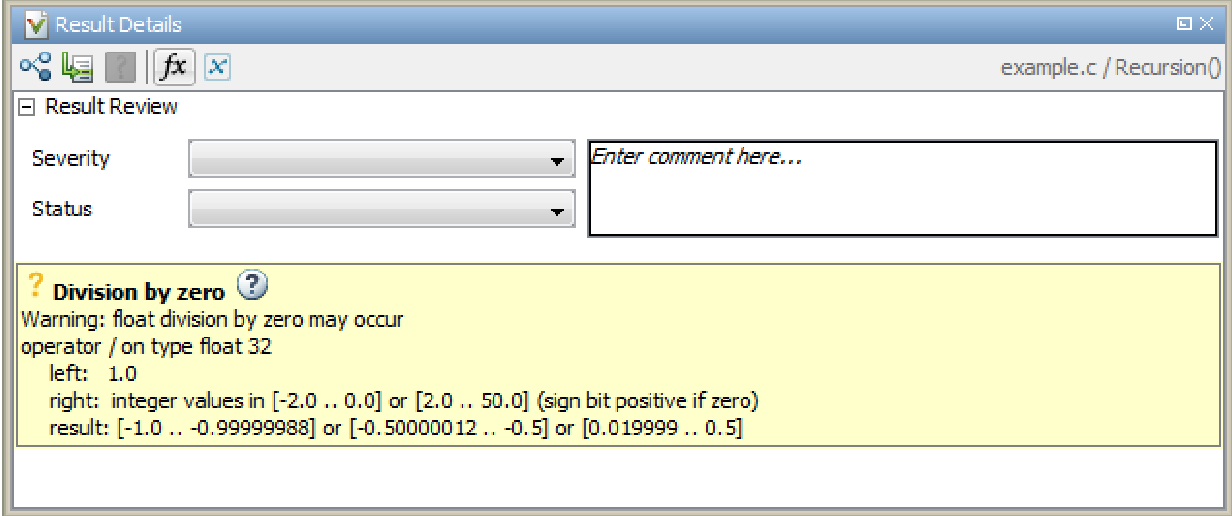
Step 1: Interpret Check Information

1. Select a check on the **Results List** pane.

   - On the **Result Details** pane, view further information about the check.
   - On the **Source** pane, the operation containing the check is highlighted.

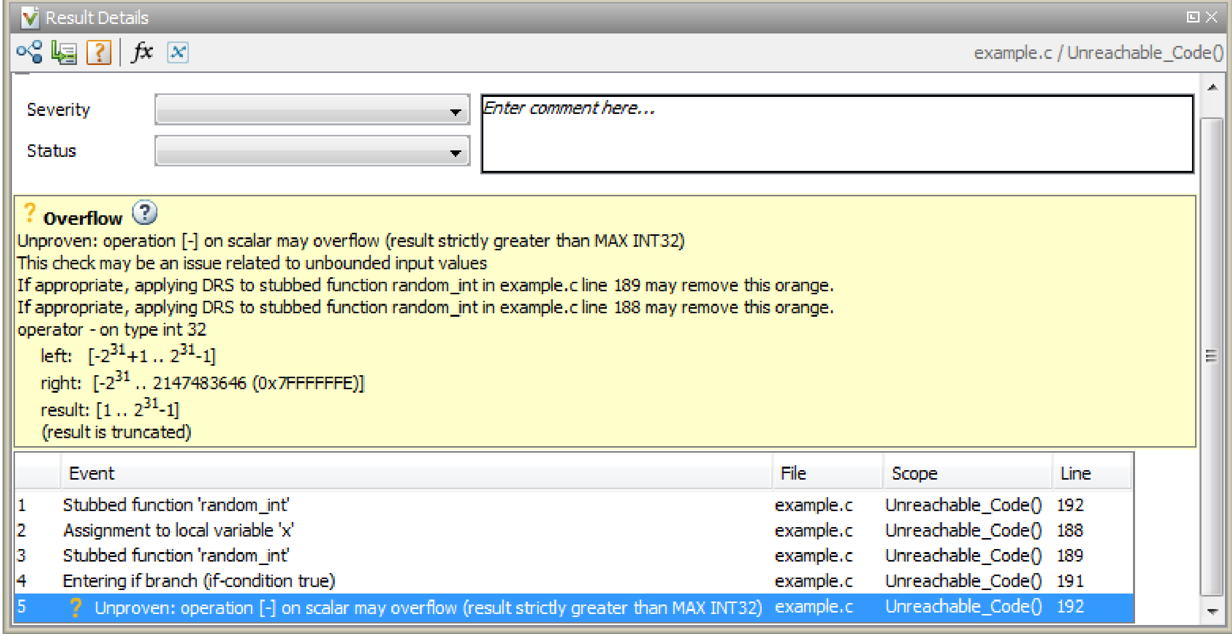
     If you place your cursor on the operation, the tooltip provides further information about the check.

   Sometimes, this information is sufficient to understand the root cause of the check. If you can determine a fix for your code from this information, you do not have to proceed further with this procedure.



For details on the information available for each check type, see "Types of Run-Time Checks" on page 8-77.

2. Sometimes, the **Result Details** pane lists the sequence of instructions that led to the check. If you see this sequence, select each instruction to trace back to the root cause of the check in your source code.



3. Sometimes, the **Result Details** pane or tooltip shows you the probable cause for the check.

   - If a line number is specified for the probable cause, right-click in the **Source** pane. Select **Go To Line**. Enter the line number.
   - If the probable cause is an undefined function, click the icon. On the **Orange Sources** pane, you can specify constraints on the arguments and return values of the function.

     For more information, see "Constrain Stubbed Functions" on page 5-44.

Step 2: Determine Root Cause of Check

If you cannot determine the root cause based on the check information, using navigation shortcuts in the user interface, navigate to the root cause.

For details on the navigation process for each type of check, see "Types of Run-Time Checks" on page 8-77. The high-level workflow is:

1 Using the tooltips on variables or operations, identify the variable `var` that causes the check. For instance, for an **Out of bound array index** error, `var` can be the array index.

2 Trace the data flow for `var`.

   a Browse through the previous instances of `var`. On the **Source** pane, place your cursor on each instance of `var` to see its values.

   Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

<table>
<tr><th>Variable</th><th>How to Find Previous Instances of Variable</th></tr>
<tr><td>Local Variable</td><td>Use one of the following methods:
• Search for the variable.
i Right-click the variable. Select Search For All References.
All instances of the variable appear on the Search pane with the current instance highlighted.
ii On the Search pane, select the previous instances.
• Browse the source code.
i Double-click the variable on the Source pane.
All instances of the variable are highlighted.
ii Scroll up to find the previous instances.</td></tr>
</table>

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.	i Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. ii On this pane, select the previous instances of the variable. Write operations on the variable are indicated with ◀ and read operations with ▶.
Function return value `ret=func();`	i Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration. ii In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 9-104.

b Find the instance where `var` acquires the value that can cause the run-time error.

3 If `var` obtains values from another variable, trace the data flow for the second variable.

   Continue this process until you identify the root cause of the check.

   For details on the navigation process for each check type, see “Types of Run-Time Checks” on page 8-77.

4 For orange checks, you have additional tools that help with root cause investigation:

   - If a function is called several times and an error occurs only on certain calls, you can identify which function call caused the check.

     For a tutorial, see “Identify Function Call with Run-Time Error” on page 9-63.

- You can run dynamic tests to see if an orange check contains a run-time error.

  However, dynamic testing cannot guarantee the absence of errors. If a test failure occurs, the orange check definitely contains a run-time error. However, if a failure does not occur, the orange check can still contain a run-time error.

  For a tutorial, see "Test Orange Checks for Run-Time Errors" on page 10-19.

  The dynamic testing of orange checks does not support certain code constructs. See "Limitations of Automatic Orange Tester" on page 10-24.

Step 3: Look for Common Causes of Check

Try to methodically determine the root cause for each check. Polyspace Code Prover lists the vulnerabilities in your code. Unless you methodically address these code vulnerabilities, you can miss possible run-time errors.

If you are aware of typical coding errors that cause a red or orange check, root cause investigation is often easier. You can browse through your source code to look for those errors.

For details on the common coding errors for each check type, see "Types of Run-Time Checks" on page 8-77.

Step 4: Trace Check to Polyspace Assumption

If you cannot determine a coding error, try to trace the check to a Polyspace assumption earlier in the code. For a list of assumptions, see "Code Prover Analysis Assumptions".

If the assumption is broader than what you expect, do one of the following:

- If you can use an analysis option to relax the assumption, rerun verification using that option.

  In particular, determine if you must specify constraints outside your code or provide other contextual information. See "Provide Context for Verification" on page 10-16.
- See if you can improve your coding design to avoid the assumption.

  For instance, `goto` statements interrupt the flow and can cause orange checks during verification. Avoid `goto` statements in your code.

To improve your coding design:

- Enforce limits on code complexity metrics. See “Review Code Metrics” on page 8-24.
- Observe coding rules. See “Follow Coding Rules” on page 10-17.

- If possible, try running verification with higher precision. See “Improve Verification Precision” on page 10-17.
- Ignore the orange check. Add a comment and justification in your result or code describing why you ignored the check. See “Add Review Comments to Results” on page 8-32 or “Add Review Comments to Code” on page 8-44.

See Also

Related Examples

- “Review Red Checks” on page 8-10
- “Review Gray Checks” on page 8-15
- “Review Coding Rule Violations” on page 12-13
- “Review Code Metrics” on page 8-24
- “Review Global Variable Usage” on page 8-29

More About

- “Results Management”
- “Result Views in Polyspace User Interface” on page 8-83

Review Code 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 verification, do the following:

- **User interface**: On the **Configuration** pane, select **Coding Rules & Code Metrics**. Select **Calculate Code Metrics**.
- **Command line**: Use the option `-code-metrics` with the `polyspace-code-prover-nodesktop` command.

After verification, the software displays code complexity metrics on the **Results List** pane. You can:

- Specify limits for the metric values through **Tools** > **Preferences**.

  If you impose limits on metrics, the **Results List** pane displays only those metric values that violate the limits. Use predefined limits or assign your own limits. If you assign your own limits, you can share the limits file to enforce coding standards in your organization.

  See “Impose Limits on Metrics” on page 8-24.
- Justify the value of a metric.

  If a metric value exceeds specified limits and appears red, you can add a comment with the rationale.

  See “Comment and Justify Limit Violations” on page 8-27.

Impose Limits on Metrics

1. Select **Tools** > **Preferences**.
2. On the **Review Scope** tab, do one of the following:

   - To use predefined limits, select **Include Quality Objectives Scopes**.

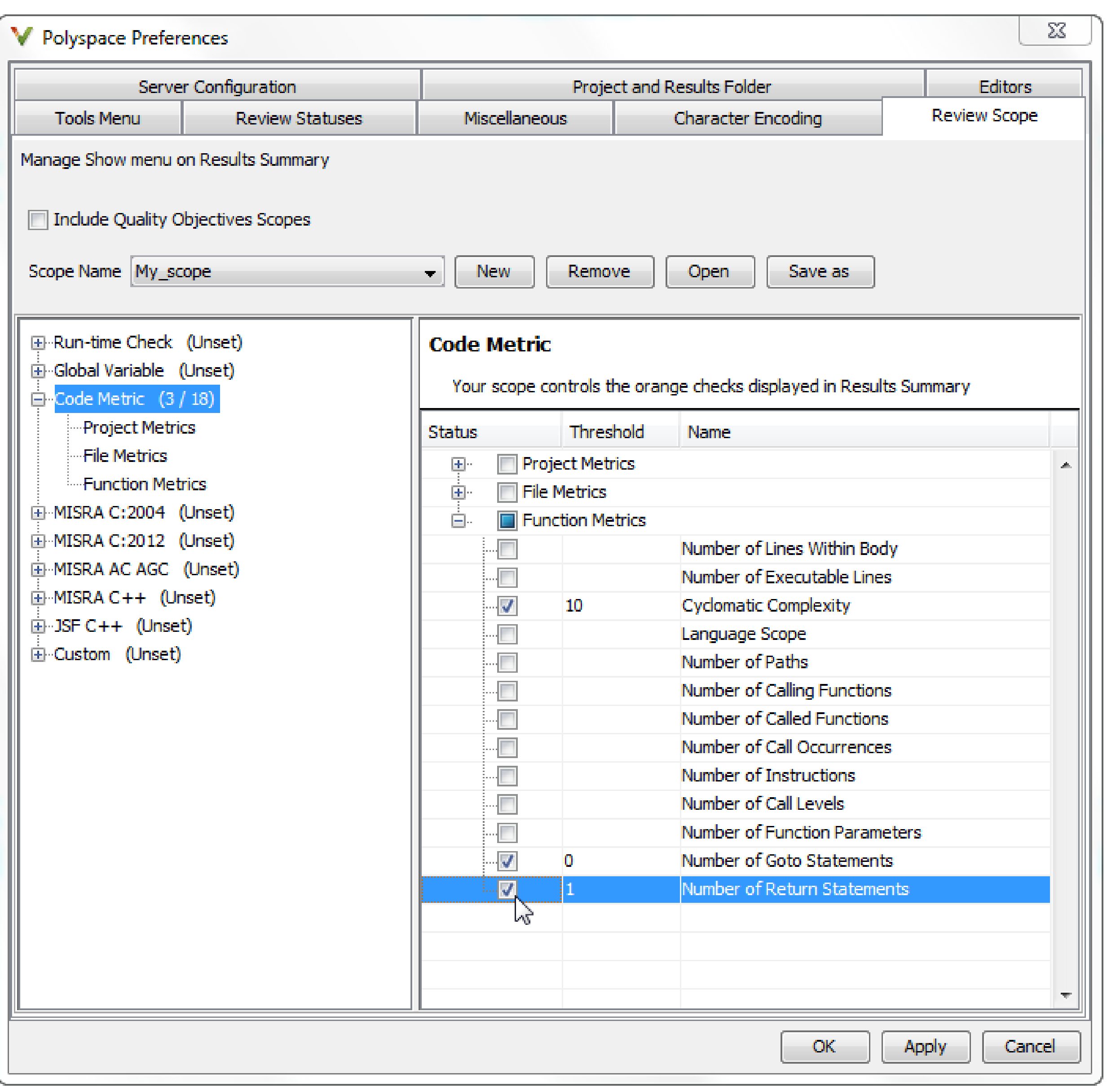
     The **Scope Name** list shows additional options, `HIS`, `SQO-4`, `SQO-5` and `SQO-6`. Select an option to see the limit values.

All the options impose the same limits on code metrics. The option `HIS` displays the HIS code metrics on page 8-81 only. The other options display other results too and impose different limits on display of orange checks. For more information, see “Limit Display of Orange Checks” on page 10-10. For a detailed explanation of the predefined limits, see “Software Quality Objectives” on page 8-122.

- 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 fraction of 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 options `HIS`, `SQO-4`, `SQO-5` and `SQO-6` appear.
- 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.

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.

Comment and Justify Limit Violations

Once you display only metrics that violate limits, you can review each violation.

1 On the **Results List** pane, from the list, select **Family**.

The code metrics appear together under one node.

2 Expand the node. Select each violation.

- On the **Results List** pane, in the **Information** column, you can see the metric value.
- On the **Result Details** pane, you can see the metric value and a brief description of the metric.

For more detailed descriptions and examples, select the icon.

3 On the **Results List** pane, add a comment and justification describing why the violation occurs. For more information, see “Add Review Comments to Results” on page 8-32.

See Also

Related Examples

- “Review Red Checks” on page 8-10
- “Review Gray Checks” on page 8-15
- “Review Orange Checks” on page 8-17
- “Review Coding Rule Violations” on page 12-13
- “Review Global Variable Usage” on page 8-29

More About

- “Results Management”
- “Result Views in Polyspace User Interface” on page 8-83

Review Global Variable Usage

After verification, Polyspace Code Prover displays a list of global variables in your source code. Using this list:

- You can remove variables that you define but do not use.

  Such variables appear gray on the **Results List** and **Source** pane.
- For code intended for multitasking, you can see which variables are not protected from concurrent access by multiple tasks.

  - If Polyspace Code Prover proves that a variable is protected, it appears green on the **Results List** and **Source** pane.
  - Otherwise, it appears orange.

For more information, see “Global Variables”.

To review global variable usage:

1. On the **Results List** pane, from the list, select **Family**.

   The global variables appear together under one node.
2. Expand the **Global Variable** node. Review each result under the nodes:

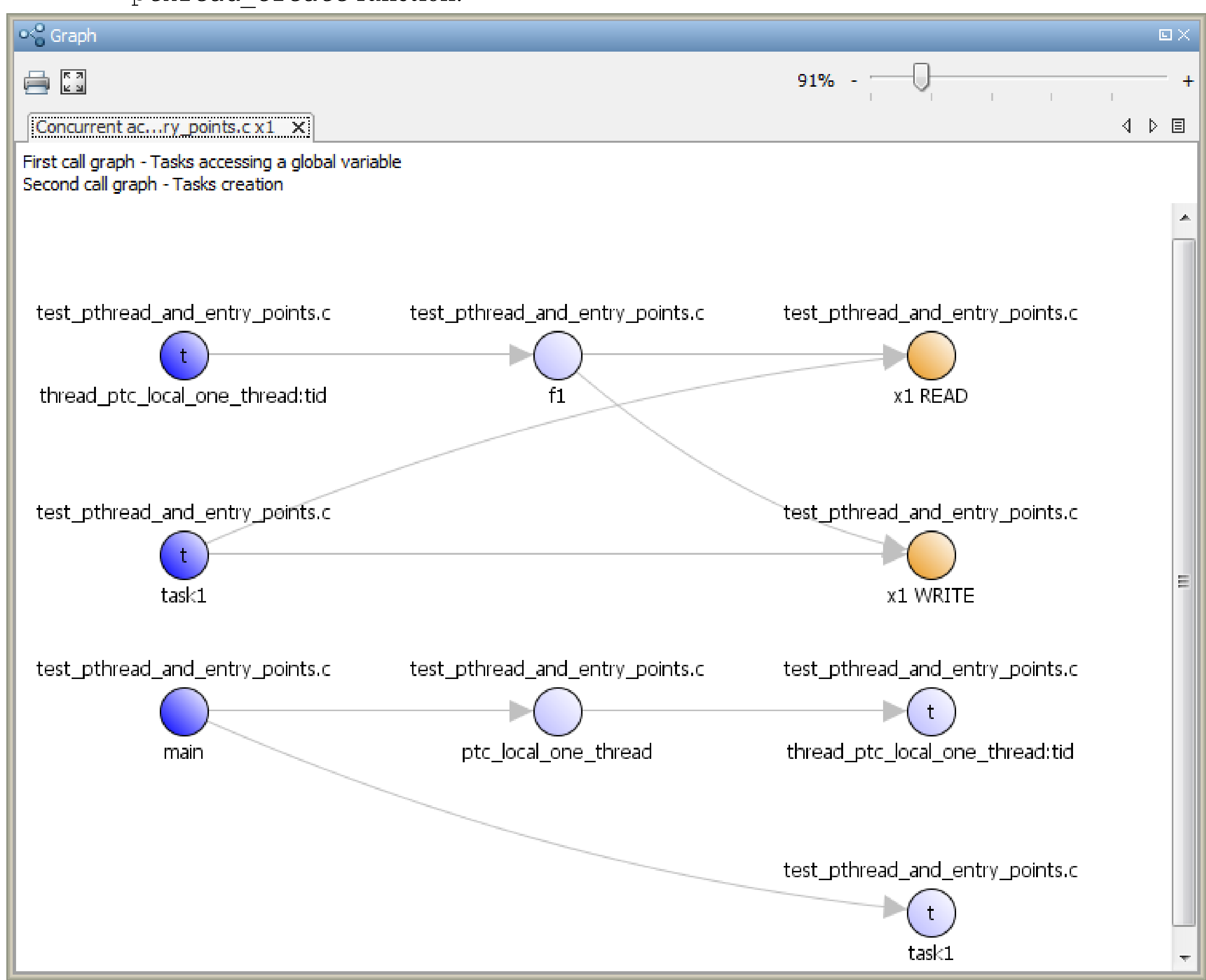
   - **Shared** > **Potentially unprotected variable**.
   - **Not shared** > **Unused variable**.
3. For each potentially unprotected variable, select the variable name.

   a. On the **Result Details** pane, view which tasks can access the variable.
   b. The read and write operations on the variable appear on this pane. Select each operation to navigate to it in your source code.

      This action also displays more details of the operation on the **Variable Access** pane.
   c. On the **Result Details** pane, click the button for a visual representation of the call sequence leading to the read and write operations.

      The following call graph shows the call sequence leading to read and write operations on variable `x1`. The call sequence begins from tasks

`thread_ptc_local_one_thread:tid` and `task1`. A second call graph shows the call sequence leading to the creation of these tasks. Task `task1` is created after `main`. Task `thread_ptc_local_one_thread:tid` is created in the function `ptc_local_one_thread` called from `main` using the `pthread_create` function.



4 To review your multitasking options, select the link **View configuration for results** on the **Dashboard** pane.

Identify whether you can leverage some of the existing protection mechanisms to protect your variable. For more information on multitasking verification, see "Multitasking".

See Also

Related Examples

- "Review Red Checks" on page 8-10
- "Review Gray Checks" on page 8-15
- "Review Orange Checks" on page 8-17
- "Review Coding Rule Violations" on page 12-13
- "Review Code Metrics" on page 8-24

More About

- "Results Management"
- "Variable Access" on page 8-104

Add Review Comments to Results

This example shows how to comment on results in the Polyspace user interface. When reviewing results, you can assign a status to them, and enter comments to describe the results of your review. These actions help you to track the progress of your review and avoid reviewing the same result twice.

On the **Results List** pane, you can filter out checks that you have reviewed. See "Filter and Group Results" on page 8-113.

Tip In the Polyspace user interface, you can quickly change to an arrangement of panes dedicated to reviewing results. Select **Window** > **Reset Layout** > **Results Review**.

Assign and Save Comments

1 On the **Results List** pane, select the result that you want to review.
2 Investigate the result further.

   For more information, see:

   - "Review Red Checks" on page 8-10
   - "Review Gray Checks" on page 8-15
   - "Review Orange Checks" on page 8-17
   - "Review Coding Rule Violations" on page 12-13
   - "Review Code Metrics" on page 8-24
   - "Review Global Variable Usage" on page 8-29
3 On the **Results List** or **Result Details** pane, provide the following review information for the result:

   - **Severity** to describe how critical you consider the issue.
   - **Status** to describe how you intend to address the issue.

     To justify the check, select one of the **Status** options, `Justified`, `No action planned` or `Not a defect`. You can view the percentage of results justified per file and function. On the **Results List** pane, from the list, select **File**. View the entries on the **Justified** column.

You can also create your own status or associate justification with an existing status. Select **Tools** > **Preferences** and create or modify statuses on the **Review Statuses** tab.

- **Comment** to describe any other information about the result.

4 To provide review information for several results together, select the results. Then, provide review information for a single result.

   To select the results in a group:

   - If the results are contiguous, left-click the first result. Then **Shift**-left click the last result.

     To group certain results together, use the column headers on the **Results List** pane.
   - If the results are not contiguous, **Ctrl**-left click each result.
   - If the results belong to the same group and have the same color, right-click one result. From the context menu, select **Select All *Color Type* Results**.

     For instance, select **Select All Orange "Illegally dereferenced pointer" Results**.

5 To save your review comments, select **File** > **Save**. Your comments are saved with the verification results.

Import Review Comments from Previous Verifications

After you have reviewed verification results, you can reuse your review comments for subsequent verifications. By default, Polyspace Code Prover imports comments from the most recent verification on the module.

After you import comments, on the **Results List** pane, clicking the icon skips justified checks. Using this icon, you can browse through unreviewed checks. You can also filter the justified checks from display. See “Filter and Group Results” on page 8-113.

Import Comments from Another Verification Result

You can import comments directly from another Code Prover result.

If a result is found in both a Bug Finder and Code Prover analysis, you can comment on the Bug Finder result and import the comment to Code Prover. For instance, most coding

rule checkers are common to Bug Finder and Code Prover. You can add comments to coding rule violations in Bug Finder and import the comments to the same violations in Code Prover.

To import comments from another set of results:

1. Open the current verification results.
2. Select **Tools** > **Import Comments**.
3. Navigate to the folder containing your previous results.
4. Select the results file with extension `.pscp` and then click **Open**.

   The review comments from the previous results are imported into the current results. For more information, see “View Imported Comments That Do Not Apply” on page 8-34.

View Imported Comments That Do Not Apply

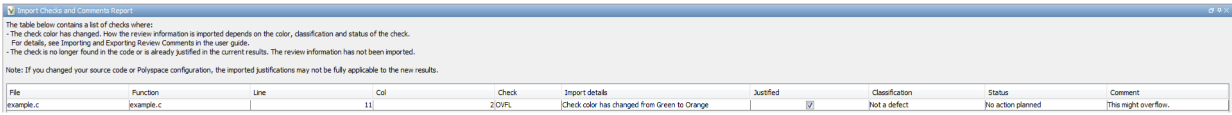
You can directly import review information from another set of results into the current results. However, it is possible that part of your review information are not imported to a subsequent verification because:

- You have changed your source code so that the check is no longer present.
- You have changed your source code so that the check color has changed.
- You have already entered different review comments for the same check.

The Import Checks and Comments Report highlights differences between two verification results. When you import comments from a previous verification, 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:

- 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 check no longer appears in the code, Polyspace highlights only the change in the Import Checks and Comments Report. It does not import review comments from the previous result.
- If you have already entered different review comments for the same check, Polyspace highlights only the change in the Import Checks and Comments Report. It does not import review comments from the previous result.

Disable Automatic Comment Import from Last Verification

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**.

See Also

Related Examples

- “Add Review Comments to Code” on page 8-44

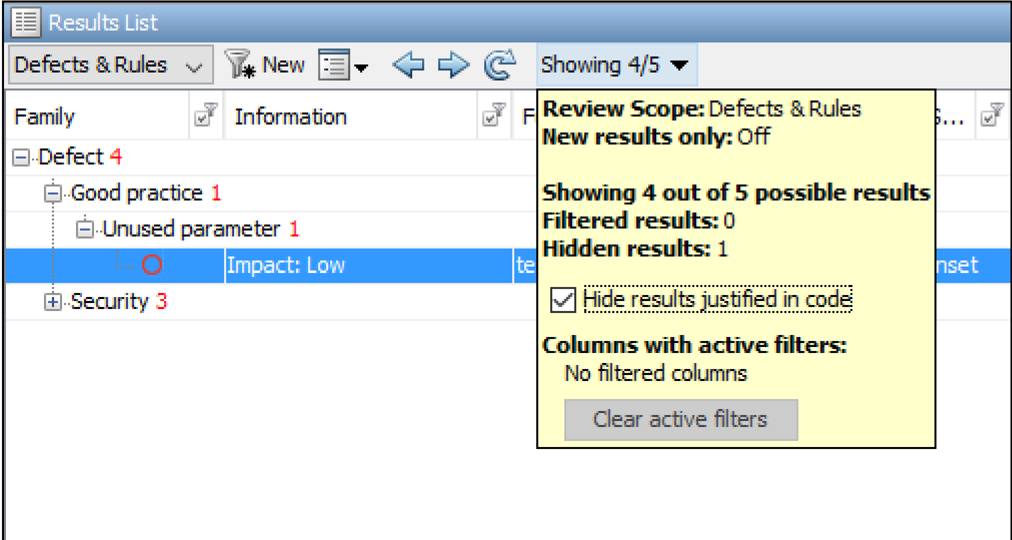
More About

- “Result Views in Polyspace User Interface” on page 8-83

Justify Results Through Code Annotations

If Polyspace finds a known or acceptable result in your code, you can hide it in subsequent analyses. Add code annotations indicating that you have reviewed the issue and you do not intend to fix it. Polyspace hides results justified through annotations in the **Results List** pane.

To make hidden results visible again, in the **Results List** pane header, click **Showing** and clear the appropriate check box.

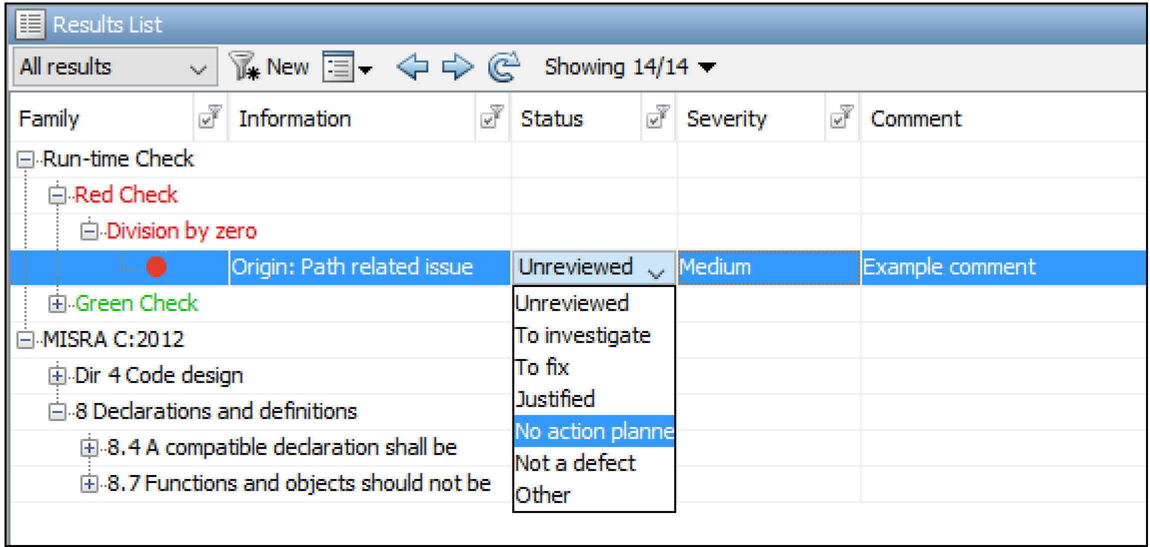


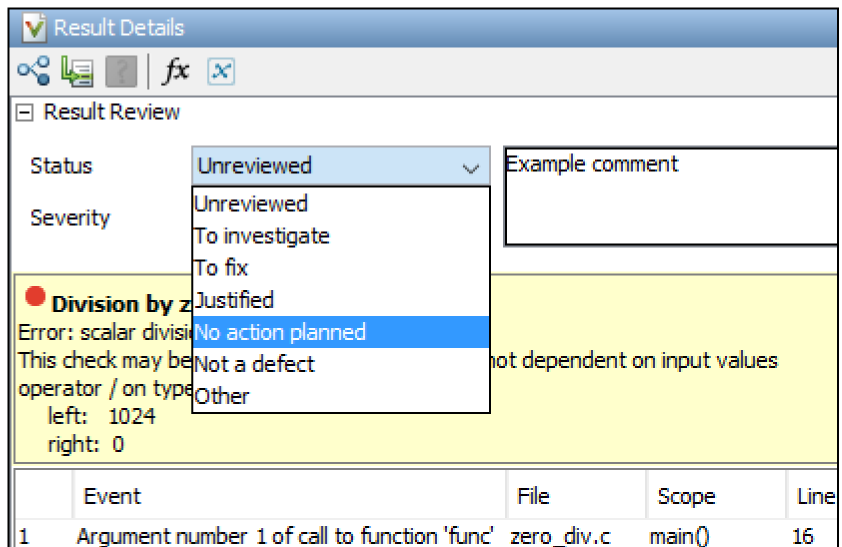
You can add annotation through the Polyspace user interface or by typing them directly in your code.

Note Results hidden through annotations still appear in generated reports.

Add Annotations from the User Interface

When you review an analysis result, you can assign a **Status** and **Severity**, and add a **Comment** from either the **Results List** or **Results Details** panes.





To convert the assigned **Status**, **Severity**, and **Comment** to a code annotation:

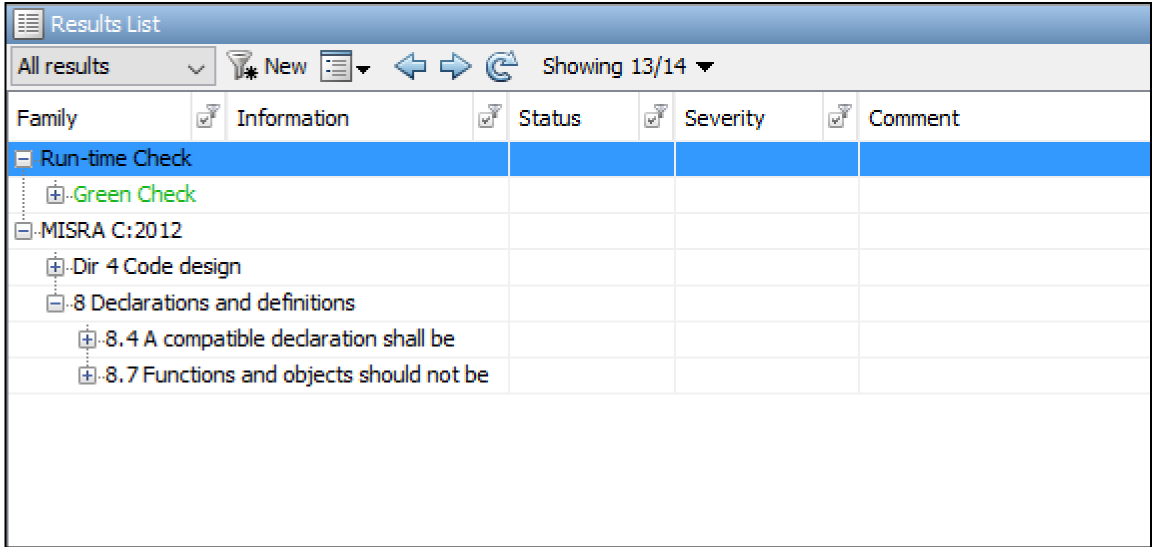
1. Right-click the result in the **Results List** pane, and then select **Add Pre-Justification to Clipboard**.

2 Right-click the result again, and then select **Open Editor**. The source file opens at the location of the defect.

3 Paste the contents of the clipboard on the line containing the defect or coding rule violation. The results **Status**, **Severity**, and **Comment** are converted to an annotation with a Polyspace syntax format.

```
int func(int p)
{
  int i;
  int j = 1;
  i = 1024 / (j - p); /*polyspace RTE:ZDV [No action planned:Medium] "Example"*/
    return i;
}
```

If you save your source file and rerun the analysis, annotated results with status `Justified`, `No action planned`, or `Not a defect` are hidden in the **Results List** pane.



Type Annotations Directly in Your Code

To add comments directly to your code, use the Polyspace annotation syntax. The syntax is not case sensitive, and has this format:

- Annotation for current line of code:

```
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"
```

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, see the values in this table or see the examples on page 8-41.

Field	Allowed Value
`Family`	Type of analysis result: • `RTE` • `VARIABLE` • `MISRA-C` or `MISRA2004` • `MISRA-C3` or `MISRA2012` • `MISRA-AC-AGC` • `JSF` • `CUSTOM` To specify all analysis results, use the asterisk character `*:*`.

Field	Allowed Value
Result_name	For RTE, use the list of acronyms for checks on page 8-69. For VARIABLE, the only allowed value is the asterisk character " * ". For coding rule violations, specify the rule number or numbers. To specify all parts of a result name [MISRA2012:17.*] or all result names in a family [RTE:*], use the asterisk character " * ".
Status	Text to indicate how you intend to address the error in your code. This value populates the **Status** column in the **Results List** pane as: • Unreviewed • To investigate • To fix • Justified • No action planned • Not a defect • Other Polyspace hides results annotated with status Justified, No action planned, or Not a defect in subsequent analyses. If you specify a status that is not an allowed value, Polyspace stores it as a custom status.

Field	Allowed Value
Severity	Text to indicate how critical you consider the error in your code. This value populates the **Severity** column in the **Results List** pane as: • Unset • High • Medium • Low If you specify a severity that is not an allowed value, Polyspace appends it to the status field and stores it as a custom status. For example, [To investigate:sporadic] is displayed in the **Status** column of the **Results List** pane as To investigate sporadic.
Comment	Additional text, such as a keyword or an explanation for the status and severity. This value populates the **Comment** column in the **Results List** pane.

Syntax Examples

Hide a Single Result

Enter an annotation on the same line as the result and specify the *Family* (RTE) and the *Result_name* (ZDV). When you do not specify a status, Polyspace assigns the status No action planned, and then hides the result in subsequent analyses.

```
code;/* polyspace RTE:zdv */
```

Hide 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;
```

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 global variable results.

```
some code; // polyspace MISRA2012:17.* VARIABLE:*
```

Specify Multiple Result Names in the Same Annotation

After you specify the *`Family`* (`MISRA2012`), enter each *`Result_name`* separated by a comma.

```
  code; /* polyspace MISRA2012:8.4, 8.7 */
```

Add Explanatory Comments to Annotation

After you specify a *`Family`* and *`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




//Multiple comments incorrect syntax:

code; /* polyspace RTE:* "OK RTE" MISRA2004:5.2 "OK MISRA" */

//Multiple comments correct syntax:
code; /* polyspace RTE:* "OK RTE" 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 8-38 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 RTE:* [To fix] "some comment" */

//Set Status 'To fix' and Severity 'High'
code; /* polyspace JSF:* [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 8-53
- “Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results” on page 8-50

Add Review Comments to Code

Note Starting R2017b, Polyspace uses a simpler annotation format. See "Justify Results Through Code Annotations" on page 8-36.

This example shows how to place review comments in your code for a particular result. If your code comments follow a particular syntax, in a later verification on the same code, Polyspace can read the comments. Using the comments, Polyspace automatically populates the **Severity**, **Status** and **Comment** fields on the **Results List** pane. After you have placed your comments in your code, you or another reviewer can avoid reviewing the same result twice.

On the **Results List** pane, you can filter out checks that you have reviewed. See "Filter and Group Results" on page 8-113.

Tip If you enter review comments in your code, the comments will appear in verification results for all subsequent verifications on that code. Therefore, use code comments for verification results for which you or someone else is not likely to change the code.

Enter Code Comments in Specific Syntax

You can 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] >
     [Additional text] */
     ```
   - For a section of code, use the following syntax.

     ```
     /* polyspace:begin<Defect:Kind1[,Kind2] : [Severity] : [Status] >
     [Additional text] */
     ```

```
... 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 Example: Run-time Checks" on page 8-47.

Replace	Replace with
Type	**Runtime errors:** `RTE` If you run Polyspace Bug Finder on the code, this code annotation is ignored.
	Coding rule violations: • `MISRA-C` • `MISRA-AC-AGC` • `MISRA-C3` • `MISRA-CPP` • `JSF` • `Custom`
	Global variables: `VARIABLE`
Kind1,Kind2,...	**Runtime errors:** Acronyms for checks such as `ZDV`, `OVFL`, etc.. If you want the comment to apply to all checks on the following line, specify `ALL`.

Replace	Replace with
	Coding rule violations: Rule number. For more information, see "Coding Rules". If you want the comment to apply to all coding rule violations on the following line, specify `ALL`.
	Global variables: `ALL`. For global variables, the same comment syntax applies irrespective of whether they are shared or used.
`Severity`	Text that indicates how critical you consider the defect. Enter one of the following: • `Unset` • `High` • `Medium` • `Low` • `Not a defect` This text populates the **Severity** column on the **Results List** pane.

Replace	Replace with
Status	Text that indicates how you intend to correct the error in your code. Enter one of the following or any other text: • `To investigate` • `To fix` • `Justified` • `No action planned` • `Not a defect` • `Other` This text populates the **Status** column on the **Results List** pane. The status is also used in Polyspace Metrics to determine whether a result is justified. To justify a result, use `Justified`, `No action planned` or `Not a defect`.
Additional text	Any text. This text populates the **Comment** column on the **Results List** pane.

Syntax Example: Run-time Checks

- Non terminating call:

```
/* polyspace<RTE: NTC : Low : No Action Planned > Known issue */
func_containing_error();
```

- Division by zero:

```
/* polyspace<RTE: ZDV : High : To Fix > Denominator cannot be zero */
y=1/x;
```

Syntax Example: Coding Rule Violations

MISRA C:2012 rule violation:

```
/* polyspace<MISRA-C3: 13.1 : Low : Justified> Known issue */
int arr[2] = {x++,y};
```

Syntax Example: Global Variables

Unused global variable:

```
/* polyspace<VARIABLE: ALL : Low : Justified> Variable to use later*/
int var_unused;
```

Copy Comment Syntax from Polyspace User Interface

Instead of manually entering the comment in a specific syntax, you can copy the comment syntax from the Polyspace user interface and paste in your code.

1 On the **Results List** or **Result Details** pane, assign a **Severity**, **Status** and **Comment** to a result.

   a Select the result.

   b Select options from the **Severity** and **Status** dropdown lists.

   c In the **Comment** field, enter a comment that helps you recognize the result easily.

2 Copy the **Severity**, **Status** and **Comment**.

   a On the **Results List** pane, right-click the result.

   b Select **Add Pre-Justification to Clipboard**. The software copies the justification string to the clipboard.

3 Paste the **Severity**, **Status** and **Comment** in your source code.

   a On the **Results List** pane, right-click the result and select **Open Editor**.

   Your source file opens on the **Code Editor** pane or an external text editor depending on your **Preferences**. The current line is the line containing the result.

   b Using the paste option in the text editor, paste the justification template string on the line immediately before the line containing the result.

   You can see your **Severity**, **Status** and **Comment** as a code comment in a format that Polyspace can read later.

   c Save your source file.

4 Run the verification again. Open your results.

On the **Results List** pane, the software populates the **Severity**, **Status** and **Comment** fields for the result. You can either ignore these findings, or filter them from the **Results List** pane. For more information on filtering, see "Filter and Group Results" on page 8-113.

See Also

Related Examples

- "Add Review Comments to Results" on page 8-32

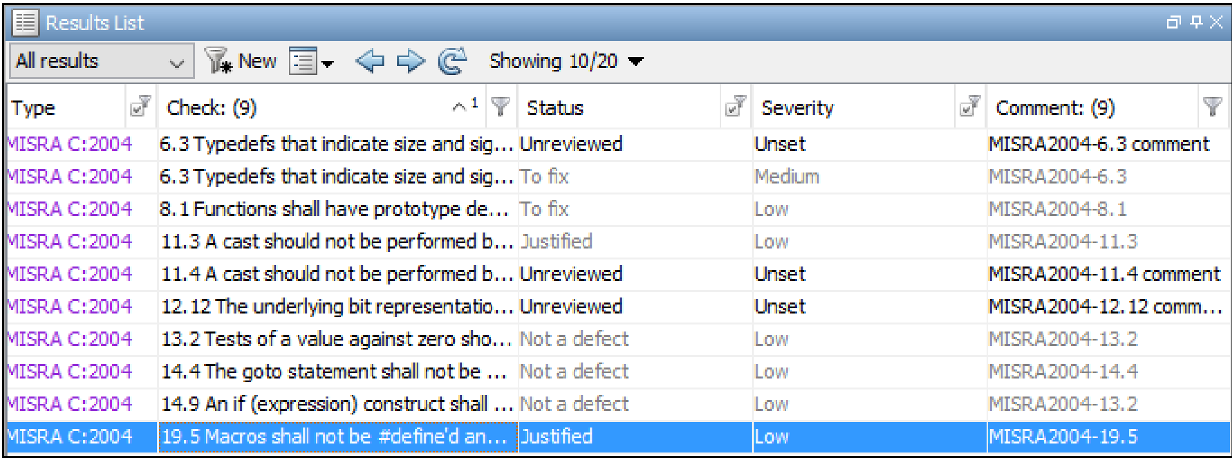
More About

- "Result Views in Polyspace User Interface" on page 8-83

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.

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.



If you are transitioning from MISRA C: 2004 to MISRA C: 2012, you do not have to review results that you have already justified.

Results List

All results | New | Showing 8/14

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

- "Justify Results Through Code Annotations" on page 8-36

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*-*)*"
Rule_Identifier_Position="1"
```

```
Status_Position="4"
Severity_Position="6"
Comment_Position="8"
               />
<!  -- Put the Regex on a single line instead of two line
when you copy it to a text editor -->

    <!-- SAME_LINE example with more complex Regex.
        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 (regex) 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 regex. 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

<table>
<tr><th>Attribute</th><th>Use</th><th>Value</th><th>Example</th></tr>
<tr><td></td><td></td><td><code>GOTO_INCREMENT</code></td><td>Applies on the same line as the annotation and the following n lines:
<pre>3. code; // myKeyword +3 ALL_MISRA
4. /*commments */
5.
6. code;
7. code;</pre>The preceding annotation applies to lines 3–6 only.</td></tr>
<tr><td></td><td></td><td><code>BEGIN</code></td><td>Applies to the same line and all following lines until a corresponding expression with attribute <code>Mode="END"</code> or <code>"END_ALL"</code>, or until the end of the file.
<pre>//myKeyword 50, 51 Block_on
Code block 1;
...</pre></td></tr>
<tr><td></td><td></td><td><code>END</code></td><td>Stops the application of a rule identifier declared by a corresponding expression with attribute <code>Mode="BEGIN"</code>.
<pre>//myKeyword 50, 51 Block_on
Code block 1;
...
More code;
//myKeyword 50 Block_off</pre>Only rule identifier 50 is turned off. Rule identifier 51 still applies.</td></tr>
</table>

Attribute	Use	Value	Example

		END_ALL	Stops all rule identifiers declared by an expression with attribute Mode="BEGIN".
			
			`//myKeyword 50, 51 Block_on`
			`Code block 1;`
			`...`
			`More code;`
			`//myKeyword Block_off_all`
			
			Rule identifiers 50 and 51 are turned off.

Regex	Required	Regular expression search string	See "Regular Expressions" (MATLAB). `Regex="myKeyword\s+(\w+(\s*,\s*\w+)*)"` matches these expressions:
			
			`// myKeyword 50, 51`
			`/* myKeyword ALL_MISRA, 100 */`

Rule_Identifier_Position	Required, except when you set Mode="END_ALL"	Integer	The integer value of this attribute corresponds to the number of opening parentheses in the regex before the relevant search expression.
			
			`<Expression Mode="GOTO_INCREMENT"`
			`Regex="myKeyword\s+(\+\d+\s)(\w+(\s*,\s*\w+`
			`Increment_Position="1"`
			`Rule_Identifier_Position="2"`
			`/>`
			
			The search expression for the rule identifier `\w+(\s*,\s*\w+)*` is after the second opening parenthesis of the regex.

Attribute	Use	Value	Example

`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 regex before the relevant search expression.
			
			`<Expression Mode="GOTO_INCREMENT"`
			`Regex="myKeyword\s+(\+\d+\s)(\w+(\s*,\s*\w+`
			`Increment_Position="1"`
			`Rule_Identifier_Position="2"`
			`/>`
			
			The search expression for the increment `\+\d+\s` is after the first opening parenthesis of the regex.

`Status_Position`	Optional	Integer	See `Increment_Position` example. When you use this attribute, the entry in your annotation is displayed in the **Status** column on the **Results List** pane of the user interface.
`Severity_Position`	Optional	Integer	See `Increment_Position` example. When you use this attribute, the entry in your annotation is displayed in the **Severity** column on the **Results List** pane of the user interface.
`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:`

Attribute	Use	Value	Example
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 annotations_description.xml
Family	Required	Corresponds to Polyspace results family. For a list of allowed values, see allowed values on page 8-38.	See the mapping section of annotations_description.xml

Attribute	Use	Value	Example
`Result_Name`	Required	Corresponds to Polyspace result names. For a list of allowed values, see allowed values on page 8-38.	See the mapping section of `annotations_description.xml`

See Also

“Annotation Description Full XML Template” on page 8-62 | `-xml-annotations-description`

More About

- “Justify Results Through Code Annotations” on page 8-36

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 8-53.

Element	Attribute	Use	Value
`Annotations`	`Group`	Required	User defined string. For example, "Custom Annotations"
`Expressions`	`Search_For_Keywords`	Required	User defined string. This string is a keyword you include in the pattern of your annotation syntax to help identify it. For example, "myKeyword"
	`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, " "

Element	Attribute	Use	Value
	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
			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.

Element	Attribute	Use	Value
	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 regex 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 regex before the relevant search expression.
	Status_Position	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regex before the relevant search expression.
	Severity_Position	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regex before the relevant search expression.

Element	Attribute	Use	Value
	`Comment_Position`	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regex 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 regex 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.

Element	Attribute	Use	Value
	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.
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 8-38.
	Result_Name	Required	Corresponds to Polyspace result names. For a list of allowed values, see allowed values on page 8-38.

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

- "Justify Results Through Code Annotations" on page 8-36

Acronyms for Checks and Code Metrics

The following table lists alphabetically the result acronyms that you must use in code comments or custom software quality objectives. For more information on the workflows, see:

- "Add Review Comments to Code" on page 8-44
- "Customize Software Quality Objectives" on page 13-25

Checks

Check	Acronym
Absolute address usage	ABS_ADDR
Correctness condition	COR
Division by zero	ZDV
Function not called	FNC
Function not reachable	FNR
Function not returning value	FRV
Illegally dereferenced pointer	IDP
Incorrect object oriented programming	OOP
Invalid C++ specific operations	CPP
Invalid operation on floats	INVALID_FLOAT_OP
Invalid shift operations	SHF
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

Check	Acronym
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

Code Complexity Metrics

You cannot add review comments to your code for code metrics. The following acronyms are useful only for defining custom software quality objectives.

Code Metric	Acronym
Comment Density	COMF
Cyclomatic Complexity	VG
Estimated Function Coupling	FCO
Higher Estimate of Local Variable Size	LOCAL_VARS_MAX
Lower Estimate of Local Variable Size	LOCAL_VARS_MIN
Language Scope	VOCF
Number of Call Levels	LEVEL
Number of Call Occurrences	NCALLS
Number of Called Functions	CALLS
Number of Calling Functions	CALLING
Number of Direct Recursions	AP_CG_DIRECT_CYCLE
Number of Executable Lines	FXLN
Number of Files	FILES
Number of Function Parameters	PARAM

Code Metric	Acronym
Number of Goto Statements	GOTO
Number of Header Files	INCLUDES
Number of Instructions	STMT
Number of Lines	TOTAL_LINES
Number of Lines Within Body	FLIN
Number of Lines Without Comment	LINES_WITHOUT_CMT
Number of Local Non-Static Variables	LOCAL_VARS
Number of Local Static Variables	LOCAL_STATIC_VARS
Number of Paths	PATH
Number of Protected Shared Variables	PSHV
Number of Recursions	AP_CG_CYCLE
Number of Return Statements	RETURN
Number of Unprotected Shared Variables	UNPSHV

Verification 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 `wrap-around` for `Overflow computation mode (-scalar-overflows-behavior)`, 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 %%

void f(int);
int read_an_input(void);

int main()  {
    int x,old_x;
    x = read_an_input();
    old_x = x;
    if (x<0 || x>10)
      return 1;
    f(x);
    x = 1 / old_x;
    // Red Division by Zero
    return 0;
}
```

```
%% file2.c %%

#include <math.h>

void f(int a) {
    int tmp;
    tmp = sqrt(0-a);
}
```

A red check occurs on `x=1/old_x;` in `file1.c` because of the following sequence of steps during verification:

1. When `x` is assigned to `old_x` in `file1.c`, the verification assumes that `x` and `old_x` have the full range of an integer, that is `[-2^31 , 2^31-1]`.
2. Following the `if` clause in `file1.c`, `x` is in `[0,10]`. Because `x` and `old_x` are equal, Polyspace considers that `old_x` is in `[0,10]` as well.
3. When `x` is passed to `f` in `file1.c`, the only possible value that `x` can have is 0. All other values lead to a run-time exception in `file2.c`, that is `tmp = sqrt(0-a);`.
4. A red error occurs on `x=1/old_x;` in `file1.c` because the software assumes `old_x` to be 0 as well.

See Also

Related Examples

- "Review Red Checks" on page 8-10
- "Review Gray Checks" on page 8-15
- "Review Orange Checks" on page 8-17

Types of 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 “Result and Source Code Colors” on page 8-3.

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 9-16
`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 9-19
`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 9-50
`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 9-54
`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 9-57
`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 9-86
`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 9-93

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 9-10
`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 9-37
`Overflow`	Review prior operations in your code that lead to an operation overflowing.	"Review and Fix Overflow Checks" on page 9-79

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 9-3
`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 9-23
`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 9-74

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 9-60

Check	How to Review	Details
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 9-65

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 9-34
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 9-21
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 9-31
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 9-72
Uncaught exception	Investigate how an exception can escape uncaught from the function where it is thrown.	"Review and Fix Uncaught Exception Checks" on page 9-90

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 9-4
`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 9-43
`User assertion`	Investigate why the condition in an `assert` statement fails.	“Review and Fix User Assertion Checks” on page 9-99

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 “Review Code Metrics” on page 8-24.

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

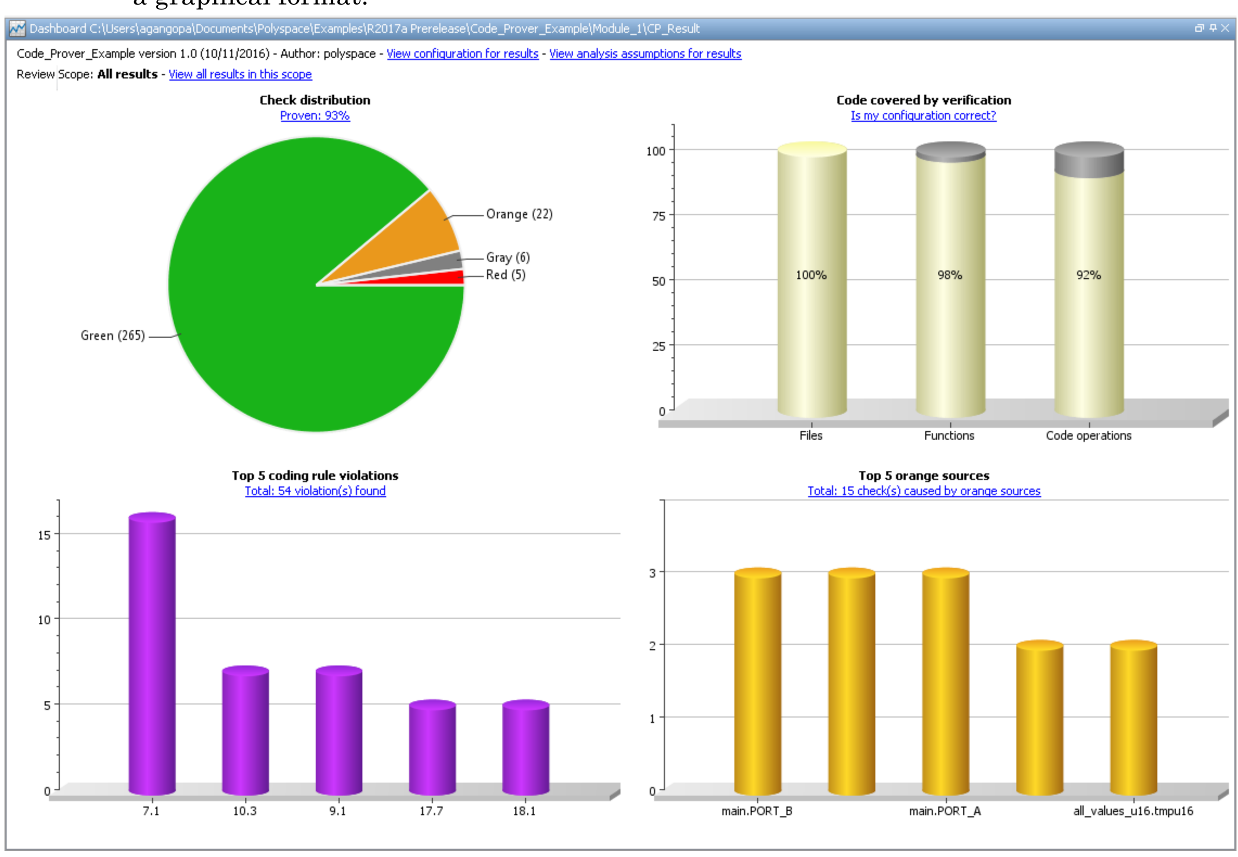
Metric	Recommended Upper Limit
`Number of Paths`	80
`Number of Return Statements`	1

Result Views in Polyspace User Interface

In the Polyspace user interface, you can use the following panes to review your results. If a pane is not open by default, to see the pane, select **Window** > **Show/Hide View** > ***Pane Name***.

Dashboard

The **Dashboard** tab on the **Source** pane provides statistics on the verification results in a graphical format.



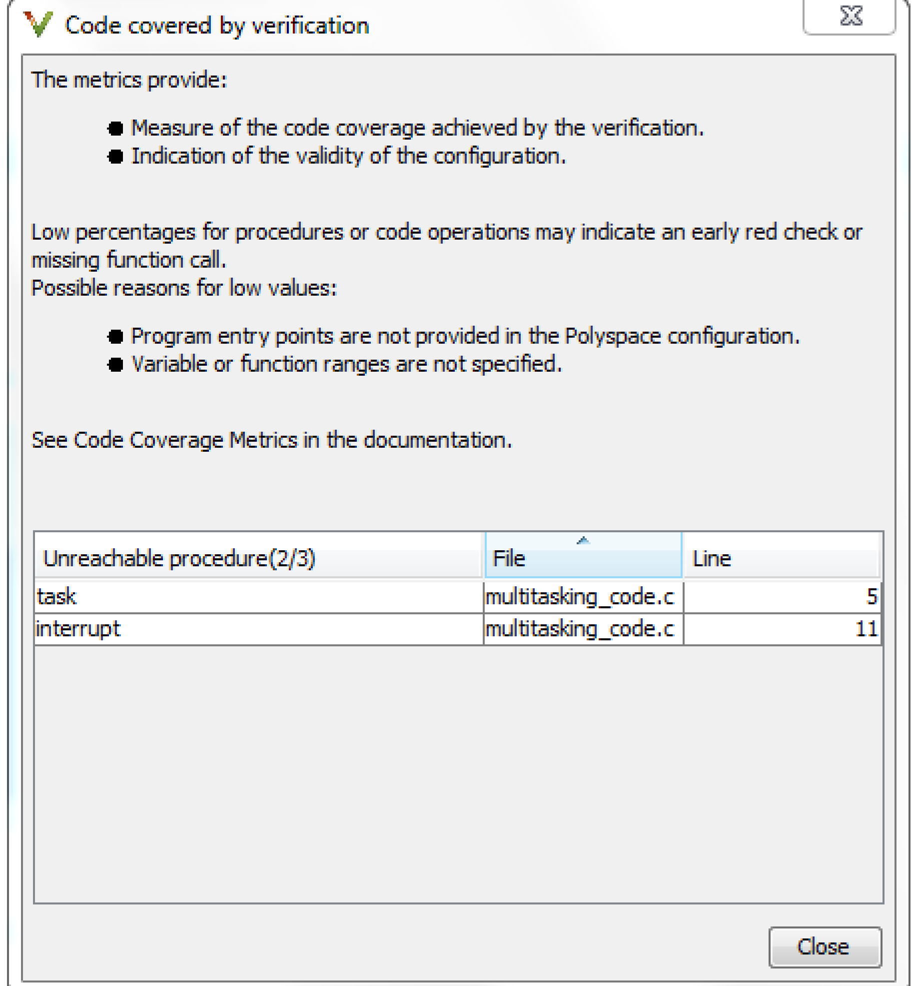
On this tab, you can view four graphs and charts:

- **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 Code covered by verification window.



This window contains:

- The fraction of procedures that are unreachable in the format, `Number of unreachable procedures/Total number of procedures`.
- A list of unreachable procedures along with the file and line number where they are defined. Selecting a procedure displays the procedure definition in the **Source** pane.

A low coverage can indicate an early red check or missing function call. Consider the following code:

```
1  void coverage_eg(void)
2  {
3    int x;
4
5    x = 1 / x;
6    x = x + 1;
7    propagate();
8  }
```

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:

- On line 5, for the division operation, a **Division by zero** check.
- On line 5, for the division operation, an **Overflow** check.
- On line 6, for the addition operation, an **Overflow** check.
- On line 6, for another read operation on x, a **Non-initialized local variable** check.

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 7-64.

- **Check distribution**

  This pie chart displays the number of checks of each color. For a description of the check colors, see "Result and Source Code Colors" on page 8-3.

  Using this pie chart, you can obtain an estimate of:

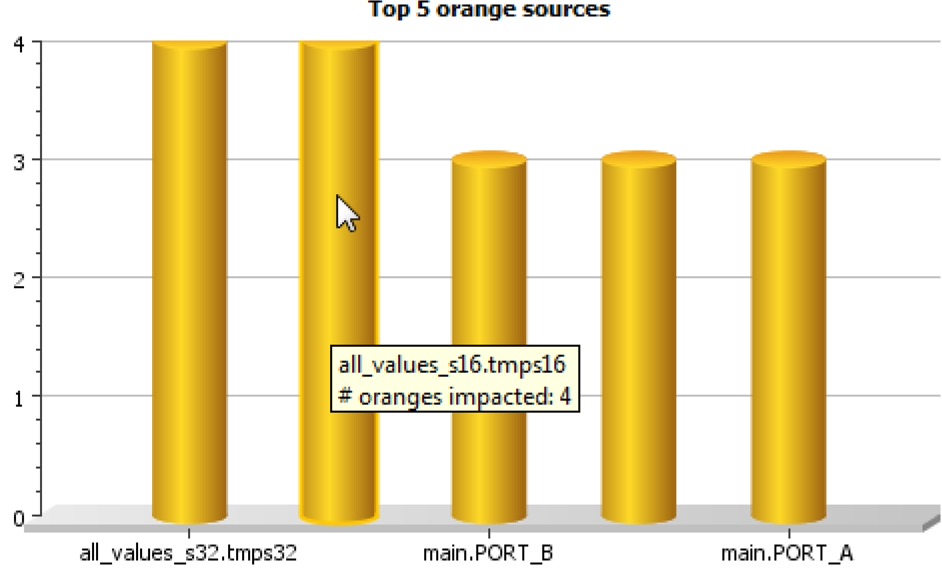
  - The number of checks to review.
  - The selectivity of your verification — the fraction of checks that are not orange.

    You can follow certain coding rules or specify certain verification options to reduce the number of orange checks. See "Reduce Orange Checks" on page 10-16.

- **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.

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 After Analysis” on page 5-36.

- 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.

For a list of supported coding rules, see “Supported MISRA C:2004 and MISRA AC AGC Rules” on page 11-14, “MISRA C:2012 Directives and Rules”, “Supported MISRA C++ Coding Rules” on page 11-92, and “Supported JSF C++ Coding Rules” on page 11-121.

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 8-113.
- View the configuration used to obtain the result. Select the link **View configuration for results**.
- View the analysis assumptions behind the result. Select the link **View analysis assumptions for results**.

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.

Attribute	Description
Group	Category of the result, for instance: • For run-time checks: Groups such as static memory, numerical, control flow, etc. • For coding rule violations: Groups defined by the coding rule standard. For instance, MISRA C: 2012 defines groups related to code constructs such as functions, pointers and arrays, etc.
Check	Result name, for instance: • For run-time checks: Check name • For coding rule violations: Coding rule number
Information	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 10-13. For coding rule violations, this column indicates whether the rule belongs to the `Required` subset. For global variables, this column contains the global variable name.
File	File containing the instruction where the result occurs
Class	Class containing the instruction where the result occurs. If the result is not inside a class definition, then this column contains the entry, **Global Scope**.

Attribute	Description
Function	Function containing the instruction where the result occurs. If the function is a method of a class, it appears in the format *class_name*::*function_name*.
Folder	Path to the folder that contains the source file with the result
Line	Line number of the instruction where the result occurs.
Col	Column number of the instruction where the result occurs. The column number is the number of characters from the beginning of the line.
%	Percentage of run-time checks that are not orange (total selectivity rate). This column is most useful when you choose the option **File** from the list. The entry in this column against a file or function indicates the percentage of checks in the file or function that are not orange.
Severity	Level of severity you have assigned to the result. The possible levels are: • `Unset` • `High` • `Medium` • `Low` • `Not a defect`

Attribute	Description
Status	Review status you have assigned to the result. The possible statuses are: • `Unreviewed` (default status) • `To investigate` • `To fix` • `Justified` • `No action planned` • `Not a defect` • `Other`
Justified	Check boxes showing whether you have justified the results. To justify a result, you must assign the status `Justified`, `No action planned` or `Not a defect`. If you choose the option **File** from the list, this column indicates the percentage of checks that you have justified per file and function.
Comments	Comments you have entered about the result

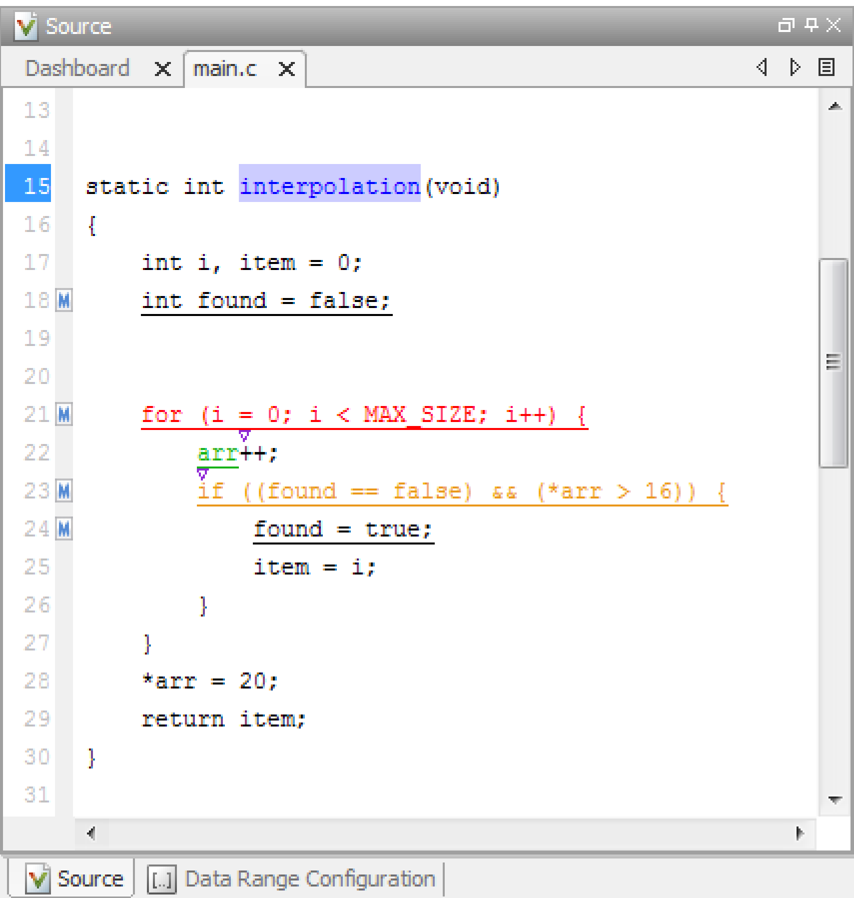
To show or hide any of the columns, right-click anywhere on the column titles. From the context menu, select or clear the title of the column that you want to show or hide.

Using this pane, you can:

- Navigate through the results. For more information, see “Add Review Comments to Results” on page 8-32.
- Organize your result review using filters on the columns. For more information, see “Filter and Group Results” on page 8-113.

Source

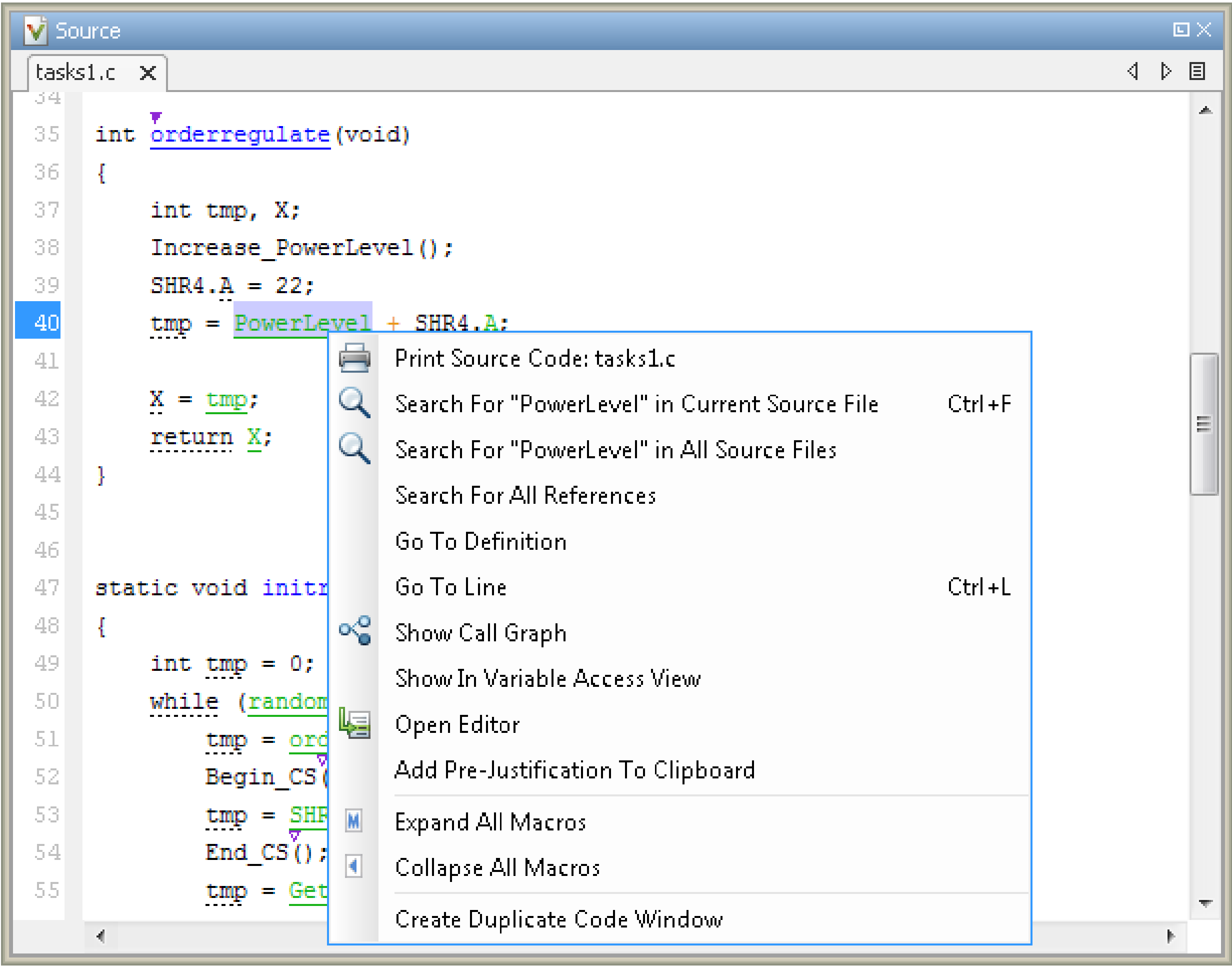
The **Source** pane shows the source code with the results highlighted with specific colors and icons. For more information, see “Result and Source Code Colors” on page 8-3.



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.

```
Source
main.c
20
21 M    for (i = 0; i < MAX_SIZE; i++) {
22          arr++;
23 M        if ((found == false) && (*arr > 16)) {
24 M            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.

```
Source
main.c
20
21      for (i = 0; i < 10; i++) {
22          arr++;
23 M        if ((found == false) && (*arr > 16)) {
24 M            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**.

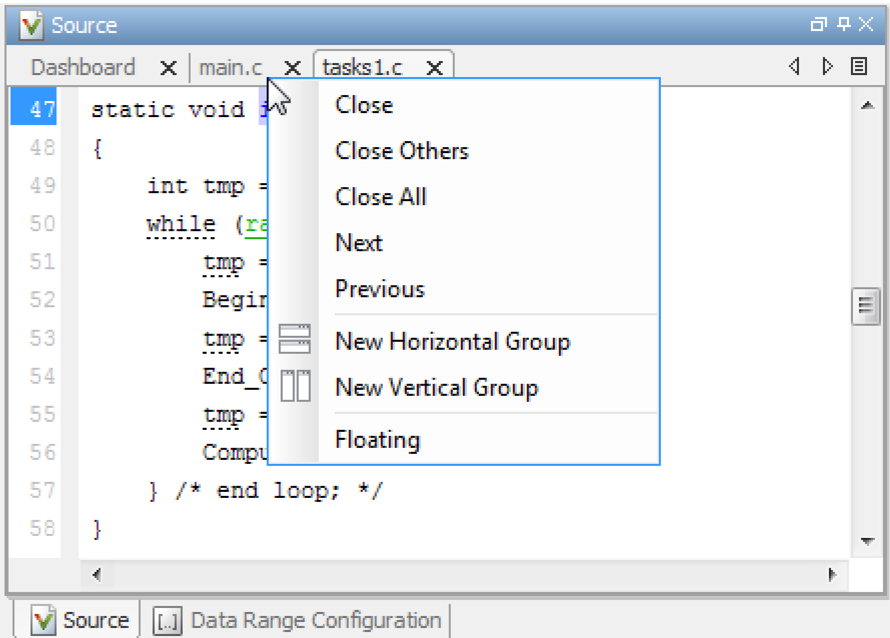
Note

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 x 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.

```
int func(int val) {
    if(val==0)
        return 0;
    return 1;
}
```

→

Call Hierarchy

Calls	Line
file.func	13
file.main	8

- When a function is called, the function call either shows a run-time check color or not. If the function does not have a run-time check color (see `func2` below), click the function name to update the **Call Hierarchy** pane.

```
int main() {
    int val = INIT_VAL;
    int checkSuccess = func (val);
    if (checkSuccess == 0)
        func2();
}
```

→

Call Hierarchy

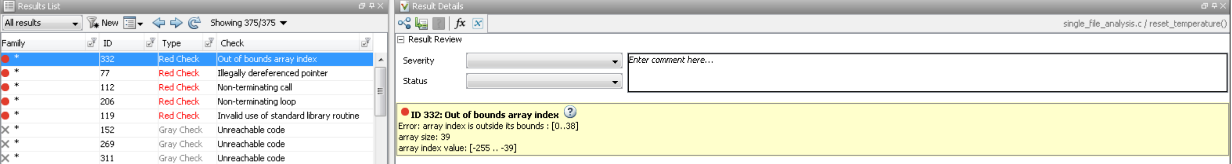
Calls	Line
file.func2	19
file.main	10

If the function has a run-time check color (see `func` above), right-click the function and select **Go To Definition**. The **Call Hierarchy** pane updates to show the callers and callees.

Result Details

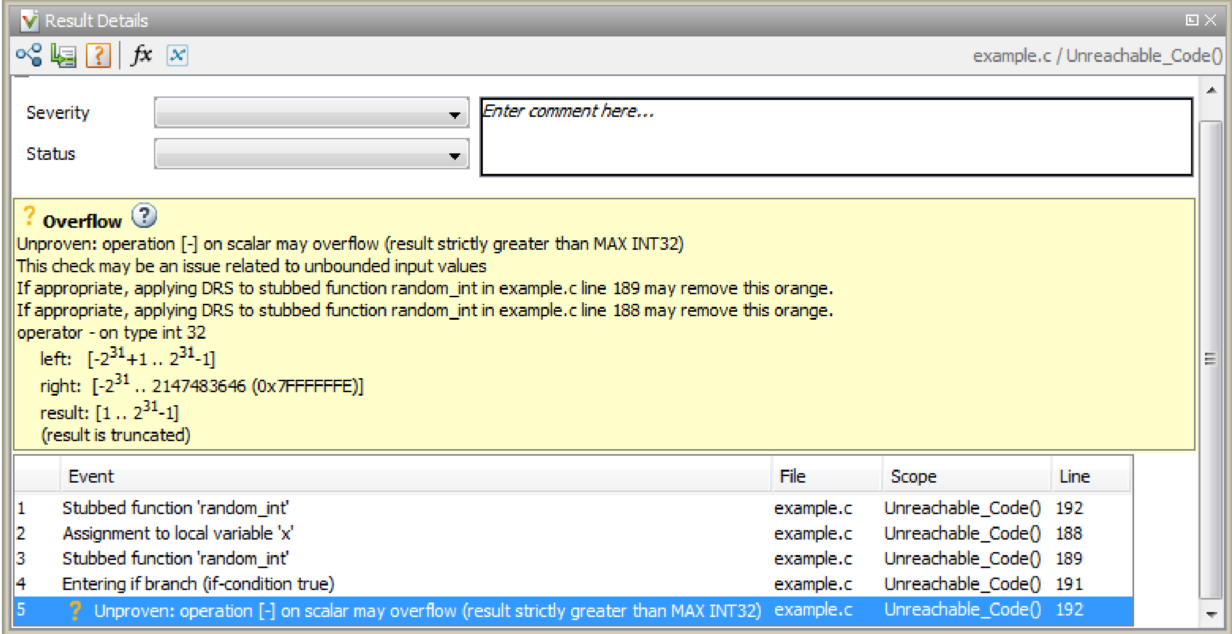
On the **Results List** pane, if you select a check, you see additional information on the **Result Details** pane.

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.



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.

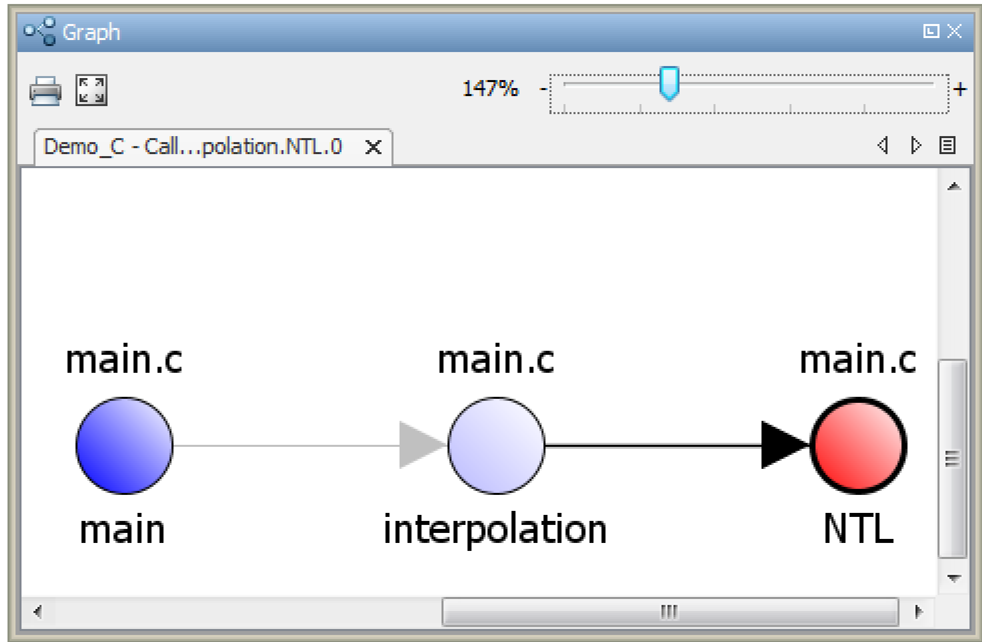


The following columns appear in the traceback:

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. For more information, see "Review Global Variable Usage" on page 8-29.

Show Call Hierarchy and Variable Access

From the **Result Details** pane, you can open the **Call Hierarchy** and **Variable Access** panes.

- Select the *fx* button to open the **Call Hierarchy** pane.

  On this pane, you can see the function in which the current check occurs, along with its callers and callees. For more information, see "Call Hierarchy" on page 8-102.
- 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 8-104.

Call Hierarchy

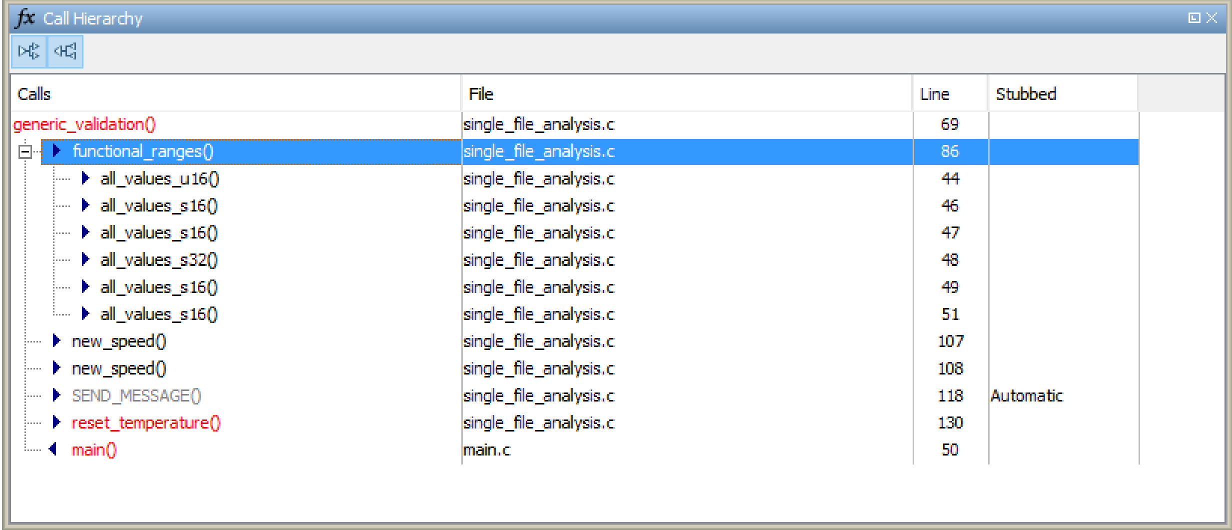
The **Call Hierarchy** pane displays the call tree of functions in the source code.

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 both 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.

You open the **Call Hierarchy** pane using the icon in your result details. 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 along 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 the following example, the **Call Hierarchy** pane displays the function, `generic_validation`, along with its callers and callees.



Depending on the name, the corresponding line number in the **Call Hierarchy** pane refers to a different line in the source code:

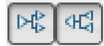
- For the function name, the line number refers to the beginning of the function definition. In the preceding example, the definition of `generic_validation` begins on line 69.
- For a callee name, the number refers to the line where the callee is called. In the preceding example, callee, `functional_ranges`, is called by `generic_validation` on line 86.
- For a caller name, the number refers to the line where the caller calls the function. In the preceding example, caller `main` calls `generic_validation` on line 50.

Tip Select a caller or callee name to navigate to the call location in the source code.

You can perform the following actions from the **Call Hierarchy** pane:

- **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 using this pane. 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**

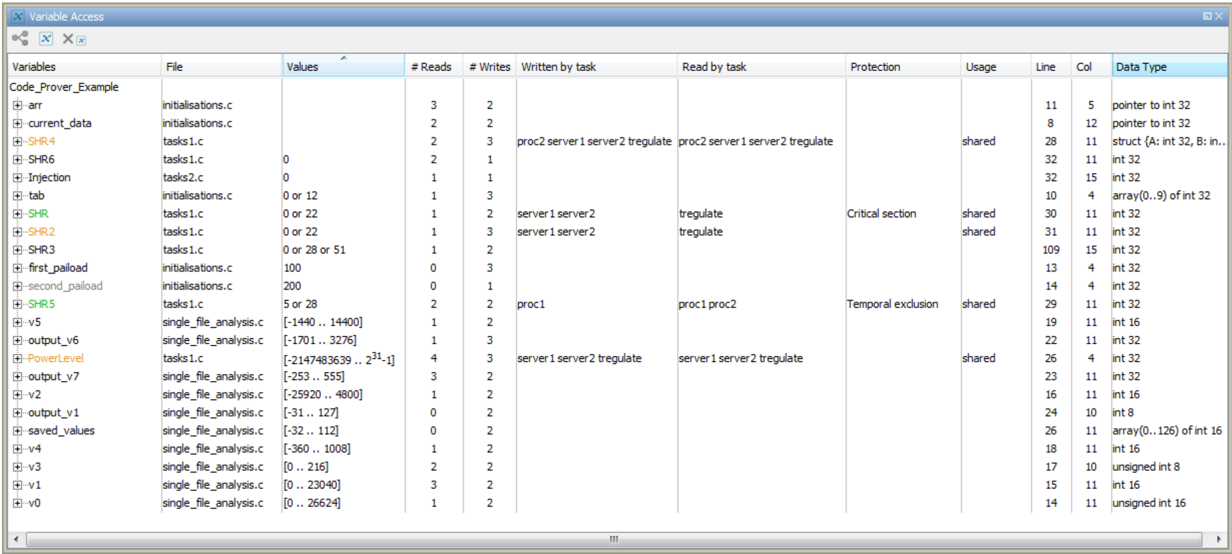
  You can determine from the **Stubbed** column if a function is stubbed. The entries in the column show why a function was stubbed.

  - **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 `-function-behavior-specifications`.

  For more information, see “Stubbed Functions”.

Variable Access

The **Variable Access** pane displays global 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.


Variable Access

Variables	File	Values	# Reads	# Writes	Written by task	Read by task	Protection	Usage	Line	Col	Data Type
Code_Prover_Example											
arr	initialisations.c		3	2					11	5	pointer to int 32
current_data	initialisations.c		2	2					8	12	pointer to int 32
SHR4	tasks1.c		2	3	proc2 server1 server2 tregulate	proc2 server1 server2 tregulate		shared	28	11	struct {A: int 32, B: in..
SHR6	tasks1.c	0	2	1					32	11	int 32
Injection	tasks2.c	0	1	1					32	15	int 32
tab	initialisations.c	0 or 12	1	3					10	4	array(0..9) of int 32
SHR	tasks1.c	0 or 22	1	2	server1 server2	tregulate	Critical section	shared	30	11	int 32
SHR2	tasks1.c	0 or 22	1	3	server1 server2	tregulate		shared	31	11	int 32
SHR3	tasks1.c	0 or 28 or 51	1	2					109	15	int 32
first_paiload	initialisations.c	100	0	3					13	4	int 32
second_paiload	initialisations.c	200	0	1					14	4	int 32
SHR5	tasks1.c	5 or 28	2	2	proc1	proc1 proc2	Temporal exclusion	shared	29	11	int 32
v5	single_file_analysis.c	[-1440 .. 14400]	1	2					19	11	int 16
output_v6	single_file_analysis.c	[-1701 .. 3276]	1	3					22	11	int 32
PowerLevel	tasks1.c	[-2147483639 .. 2^{31}-1]	4	3	server1 server2 tregulate	server1 server2 tregulate		shared	26	4	int 32
output_v7	single_file_analysis.c	[-253 .. 555]	3	2					23	11	int 32
v2	single_file_analysis.c	[-25920 .. 4800]	1	2					16	11	int 16
output_v1	single_file_analysis.c	[-31 .. 127]	0	2					24	10	int 8
saved_values	single_file_analysis.c	[-32 .. 112]	0	2					26	11	array(0..126) of int 16
v4	single_file_analysis.c	[-360 .. 1008]	1	2					18	11	int 16
v3	single_file_analysis.c	[0 .. 216]	2	2					17	10	unsigned int 8
v1	single_file_analysis.c	[0 .. 23040]	3	2					15	11	int 16
v0	single_file_analysis.c	[0 .. 26624]	1	2					14	11	unsigned int 16

For each variable and each read/write access, the **Variable Access** pane contains the relevant attributes. For the variables, the various attributes are listed in this table.

Attribute	Description
Variables	Name of Variable
File	Source file containing variable declaration
Values	Value (or range of values) of variable This column is empty for pointer variables.
# Reads	Number of times the variable is read
# Writes	Number of times the variable is written
Written by task	Name of tasks writing on variable
Read by task	Name of tasks reading variable

Attribute	Description
Protection	Whether shared variable is protected from concurrent access (Filled only when **Usage** column has entry, **Shared**) The possible entries in this column are: • **Critical Section**: If variable is accessed in critical section of code • **Temporal Exclusion**: If variable is accessed in mutually exclusive tasks For more details on these entries, see "Multitasking".
Usage	`Shared`, if variable is shared between tasks; otherwise, blank
Line	Line number of variable declaration
Col	Column number (number of characters from beginning of line) of variable declaration
Data Type	Data type of variable (C/C++ data types or structures/classes)

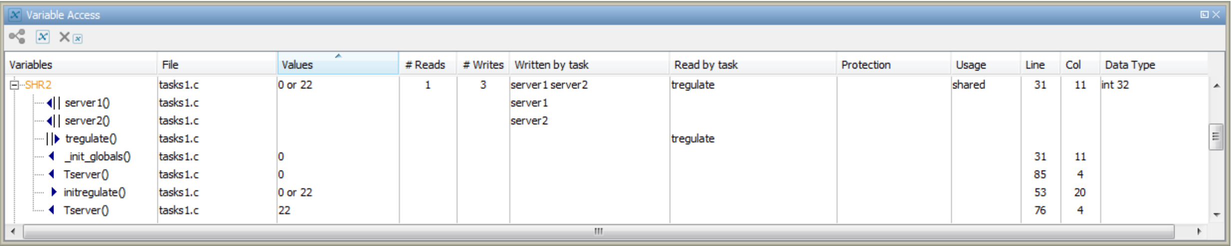
Double-click a variable name to view read/write access operations on the variable. The arrowhead symbols ▸ and ◂ in the **Variable Access** pane indicate functions performing read and write access respectively on the global variable. Likewise, tasks performing read and write access are indicated by the symbols ||▸ and ◂|| respectively. For further information on tasks, see `Entry points (-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

Attribute	Description
Values	Value or range of values of variable in the function or task performing read/write access This column is empty for pointer variables.
Written by task	*Only for tasks*: Name of task performing write access on variable
Read by task	*Only for tasks*: Name of task performing read access on variable
Line	Line number where function or task accesses variable
Col	Column number where function or task accesses variable
File	Source file containing access operation on variable

For example, consider the global variable, SHR2:



The function, Tserver, in the file, tasks1.c, performs two write operations on SHR2. This is indicated in the **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.

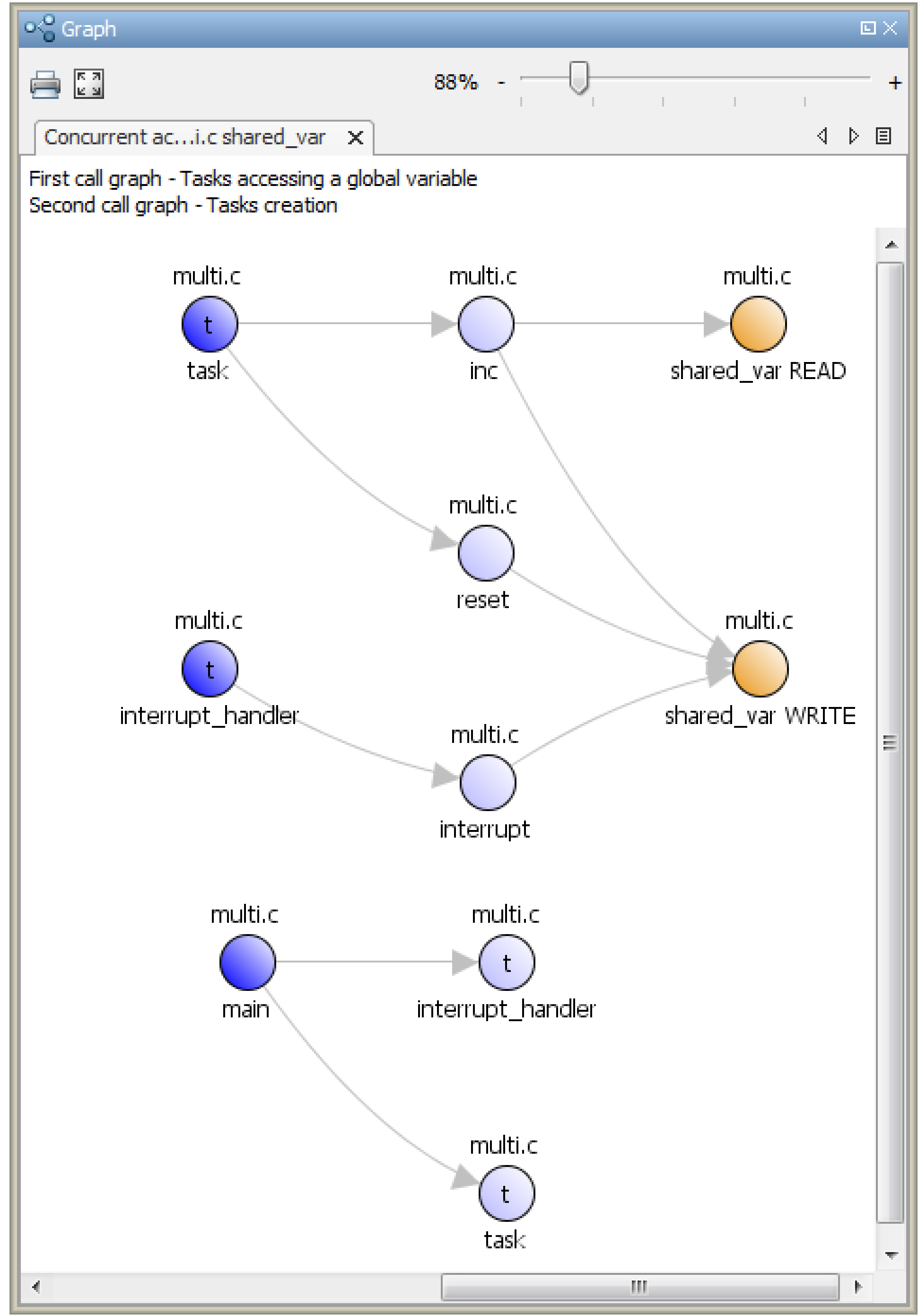
The information about global variables and read/write access operations obtained from the **Variable Access** pane is called the data dictionary.

You can also perform the following actions from the **Variable Access** pane.

- View Access Graph

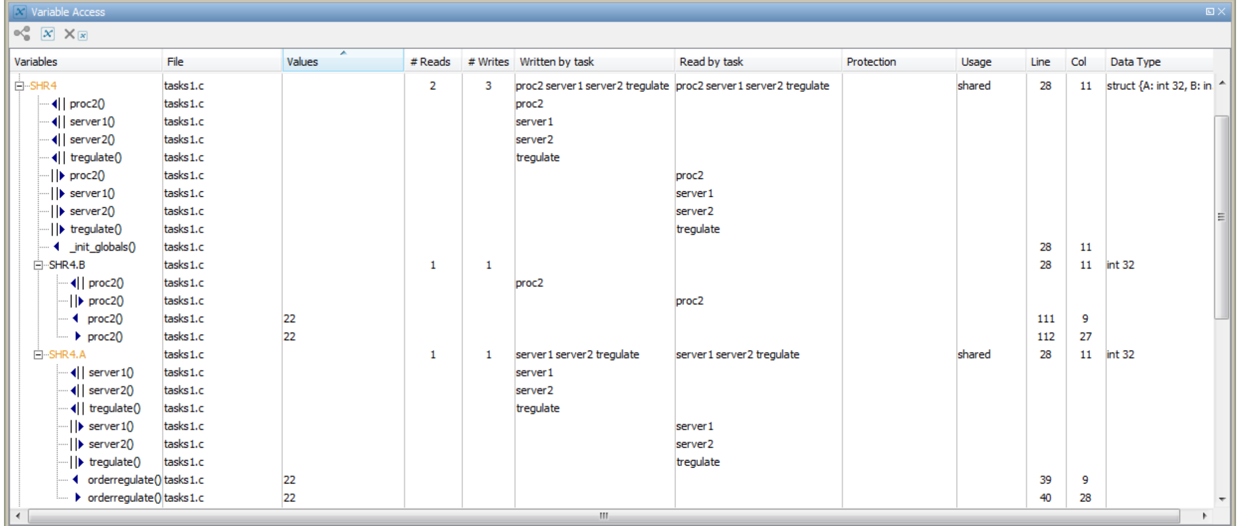
  View the access operations on a global variable in graphical format using the **Variable Access** pane. Select the global variable and click .

  Here is an example of an access graph:


Graph
88%
Concurrent ac…i.c shared_var
First call graph - Tasks accessing a global variable
Second call graph - Tasks creation
multi.c
t
task
multi.c
inc
multi.c
shared_var READ
multi.c
reset
multi.c
t
interrupt_handler
multi.c
shared_var WRITE
multi.c
interrupt
multi.c
main
multi.c
t
interrupt_handler
multi.c
t
task

- **View Structured Variables**

  For structured variables, view the individual fields from the **Variable Access** pane. For example, for the structure, `SHR4`, the pane displays the fields, `SHR4.A` and `SHR4.B`, and the functions performing read/write access on them.

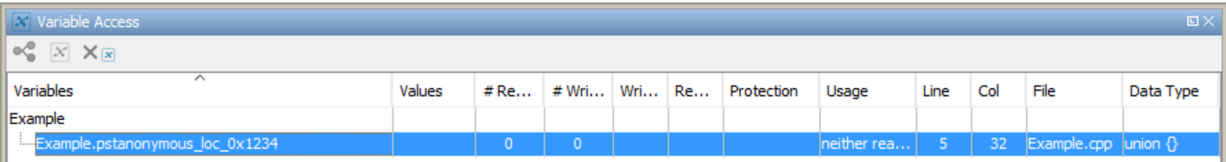


- **View Operations on Anonymous Variables**

  You can view operations on anonymous variables. For example, consider this line of code that declares an unnamed union with the variable at an absolute address:

```
union {char, c; int i; } @0x1234;
```

  When you analyze the preceding code and specify the `iar` compiler, the unnamed variable at `0x1234` appears in the **Variable Access** pane with a name that starts with **pstanonymous**.

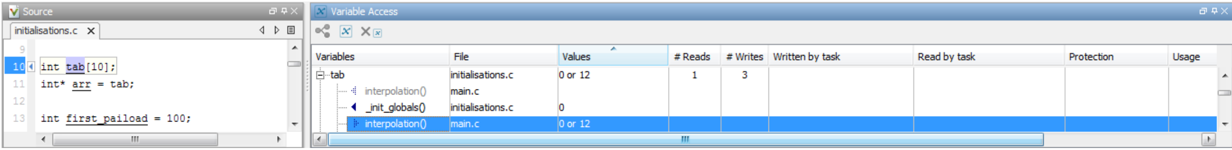


- **View Access Through Pointers**

  View access operations on global variables performed indirectly through pointers.

If a read/write access on a variable is performed through pointers, then the access is marked by (read) or (write).

For instance, in the file, `initialisations.c`, the variable, `arr`, is declared as a pointer to the array, `tab`.



In the file `main.c`, `tab` is read in the function, `interpolation()`, through the 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`.

Filter and Group Results

This example shows how to filter and group results on the **Results List** pane. To organize your result review, use filters and groups when you want to:

- Review certain types of checks in preference to others. For instance, you first want to address checks resulting from **Out of bounds array index**.
- Review only new results found since the last verification.
- Not address the full set of coding rule violations detected by the coding rules checker.
- Not review results you have already justified.

  Typically, in your second or later rounds of review, you would have some results already justified.
- Review only those results that you have already assigned a certain status. For instance, you want to review only those results to which you have assigned the status, `To investigate`.
- Review all results in the body of a particular file or function. Because of continuity of code, reviewing these results together can help you organize your review process.

  You can also review results in a file if you have written the code for that file only and not the entire set of source files used for verification.
- Not review the results in automatically generated functions.
- C++ only: Review all results dealing with a class definition.

Filter Results

You can filter results using graphs on the **Dashboard** pane or filters on the **Results List** pane. You can generate reports using only the results that are currently on display. See “Generate Report” on page 8-133.

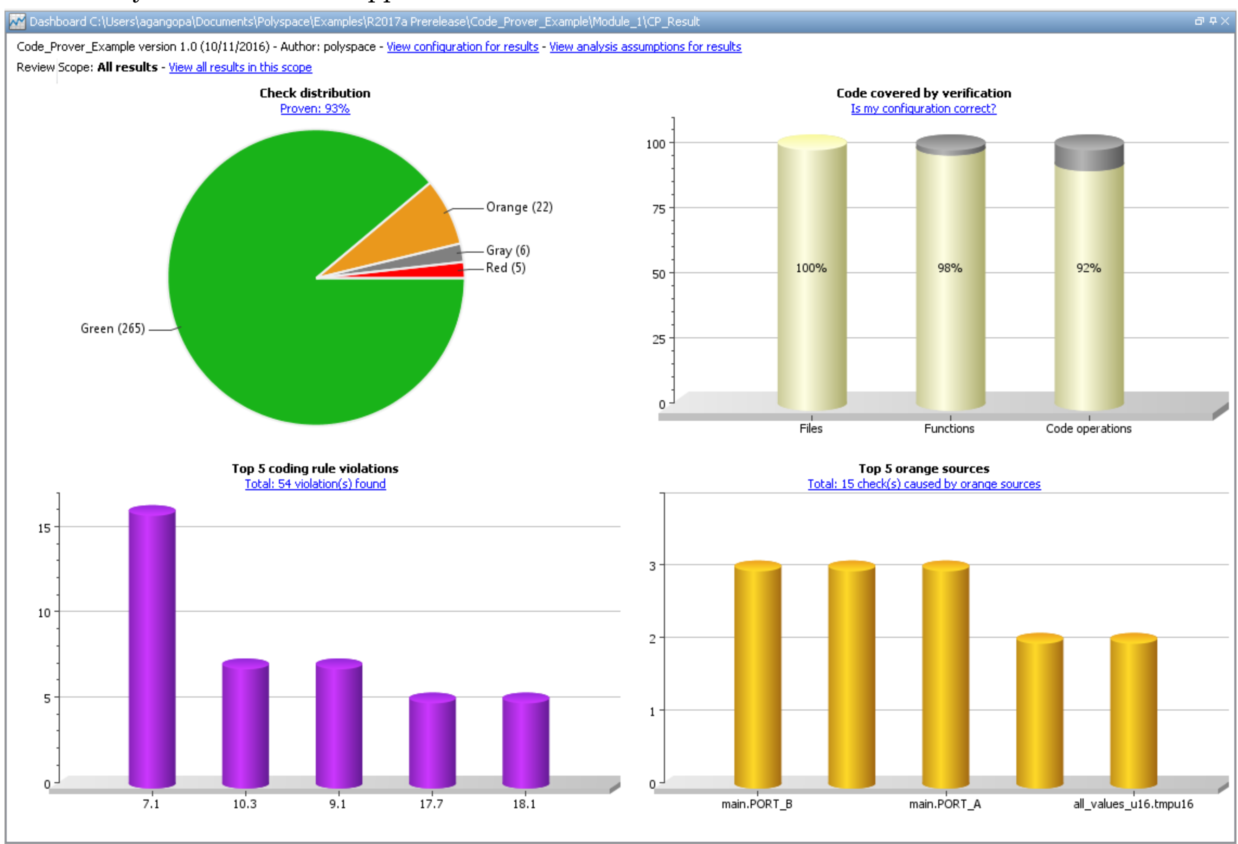
Filter Using Dashboard

The **Dashboard** pane provides a graphical overview of the results. You can click the elements on the graphs to filter results. For instance, you can use the following graphs:

- **Check distribution**: If you click a colored region on this pie chart, the **Results List** pane shows checks of that color only.

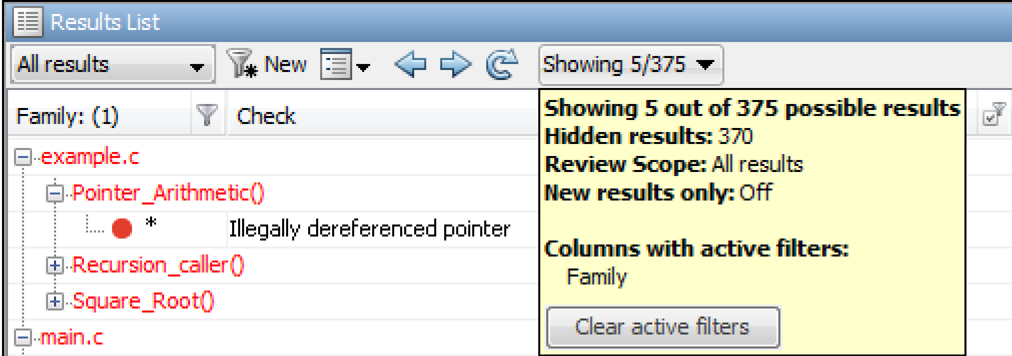
- **Top 5 coding rule violations**: If you click a column corresponding to a rule, the **Results List** pane shows violations of that rule only.
- **Top 5 orange sources**: If you click a column corresponding to an orange source, the **Results List** pane shows orange checks caused by that source only.

To clear filters from the **Dashboard** pane, select the link **View all results in this scope**. This action clears all filters and displays the available results in the scope that you choose in the upper left menu of the **Results List** toolbar.



Filter Using Results List

For other filtering mechanisms, use filters on the **Results List** pane itself. To clear filters from the **Results List** pane, use the button **Clear active filters** in the **Showing** dropdown.



- To filter results using the attributes in a certain column on the **Results List** pane, click the icon on the column. Do one of the following:
  - Clear the boxes for the results that you want filtered from display.
  - Clear **All**. Select the boxes for the results that you want displayed.

Item to Filter	Column
Results in a certain file or function	**File** or **Function**
Results in files belonging to specific folders	**Folder**
Results associated with a certain class	**Class**
Checks of a certain type, for instance, **Overflow**	**Check** The column does not appear if you group checks by family. See “Group Results” on page 8-117.
Results with a certain severity or status	**Severity** or **Status**
Results that come from the same root cause. For instance, if multiple issues trigger the same coding rule violation, you can filter on the individual issues.	**Detail**

Item to Filter	Column
Results that you have justified. If you assign the status `Justified`, `No action planned` or `Not a defect`, a result is justified.	**Justified**
Checks only	**Family**
Checks of a certain color	**Family**
Global variables of a certain type	**Family**
Code metrics	**Family**
Orange checks that are most likely run-time errors (path-related checks or checks from bounded input values)	**Information** See also "Critical Orange Checks" on page 10-13.

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.

- To review only new results found since the last verification, on the **Results List** pane, select New.
- To suppress code metrics and global variables from your results, from the drop-down list in the left of the **Results List** pane toolbar, select **Checks & Rules**.

  You can increase the options on this list or create your own options. For examples, see:

  - "Limit Display of Orange Checks" on page 10-10
  - "Suppress Certain Rules from Display in One Click" on page 12-15
  - "Review Code Metrics" on page 8-24

Note You can also apply multiple filters. Once you apply a set of filters to your verification results, they are preserved for subsequent verifications on the same project

module. The **Results List** pane shows the number of results filtered from display. If you place your cursor on the number, you can see which filters have been applied.

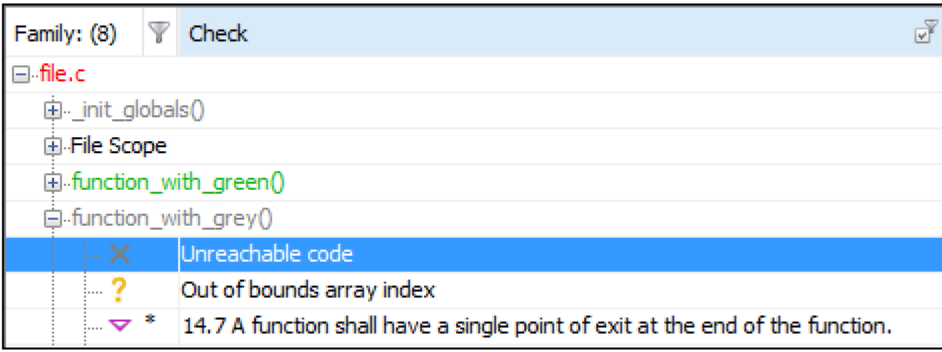
Filter Using Orange Sources

On the **Orange Sources** pane, you see the sources of imprecision, for instance, undefined (stubbed) functions and volatile variables.

If you select a source on this pane, in the **Results List** pane, you see only the orange checks caused by the source. To clear the filters, in the **Showing** dropdown on the **Results List** pane, use the button **Clear active filters**.

Group Results

On the **Results List** pane, from the list, select an option, for instance, grouping by file.



The available options are:

- **None**: Shows results without grouping.
- **Family**: Shows results grouped by result type.

  The results are organized by type: checks, global variables, coding rule violations, code metrics. Within each type, they are grouped further.

  - The checks 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 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 Rules”.
  - 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**.

  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**.

You can also click on a column header to group results. For instance, if you click on the **Detail** column header, all results that have the same detail (or same root cause) are grouped together.

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 7-64. 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 “Result Review Process”.

   Continue to click the forward arrow until you have reviewed through all of the checks.
4. Before reviewing orange checks, review red and gray checks.
5. Prioritize your orange check review by:

   - Files and functions: For easier review, begin your orange check review from files and functions with *fewer* orange checks.

To view the percentage of non-orange checks per file and function, on the **Results List** pane, from the list, select **File**. Right-click a column header and select %.

- Check type: Review orange checks in the following order. Checks are more difficult to review as you go down this order.

Review Order	Checks
First	• `Out of bounds array index` • `Non-initialized local variable` • `Division by zero` • `Invalid shift operations`
Second	• `Overflow` • `Illegally dereferenced pointer`
Third	Remaining checks

- Orange check sources: Review all orange checks caused by a single variable or function. Orange checks often arise from variables whose values cannot be determined from the code or functions that are not defined.

  To review the top sources, view the **Top 5 orange sources** graph on the **Dashboard** tab or the **Orange Sources** tab. You can also select an orange source on either tab to see only the orange checks caused by the source. For more information, see “Filter and Group Results” on page 8-113.

- 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 10-5.

8 To see what percentage of checks you have justified:

   a If you want the percentage broken down by color and type, on the **Results List** pane, from the list, select **Family**. If you want the percentage broken down by file and function, select **File**.

   b View the entries in the **Justified** column.

See Also

Related Examples

- "Review Red Checks" on page 8-10
- "Review Gray Checks" on page 8-15
- "Review Orange Checks" on page 8-17
- "Limit Display of Orange Checks" on page 10-10

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 `goto` statements	0
Cyclomatic complexity	10
Number of calling functions	5
Number of calls	7
Number of parameters per function	5
Number of instructions per function	50
Number of call levels in a function	4
Number of `return` statements in a function	1
Language scope, an indicator of the cost of maintaining or changing functions. Calculated as follows: `(N1+N2) / (n1+n2)` • *n1* — Number of different operators • *N1* — Total number of operators • *n2* — Number of different operands • *N2* — Total number of operands	4
Number of recursions	0

Metric	Threshold Value
Number of direct recursions	0
Number of unjustified violations of the following MISRA C:2004 rules: • 5.2 • 8.11, 8.12 • 11.2, 11.3 • 12.12 • 13.3, 13.4, 13.5 • 14.4, 14.7 • 16.1, 16.2, 16.7 • 17.3, 17.4, 17.5, 17.6 • 18.4 • 20.4	0
Number of unjustified violations of the following MISRA C:2012 rules: • 8.8, 8.11, and 8.13 • 11.1, 11.2, 11.4, 11.5, 11.6, and 11.7 • 14.1 and 14.2 • 15.1, 15.2, 15.3, and 15.5 • 17.1 and 17.2 • 18.3, 18.4, 18.5, and 18.6 • 19.2 • 21.3	0

Metric	Threshold Value
Number of unjustified violations of the following MISRA C++ rules: • 2-10-2 • 3-1-3, 3-3-2, 3-9-3 • 5-0-15, 5-0-18, 5-0-19, 5-2-8, 5-2-9 • 6-2-2, 6-5-1, 6-5-2, 6-5-3, 6-5-4, 6-6-1, 6-6-2, 6-6-4, 6-6-5 • 7-5-1, 7-5-2, 7-5-4 • 8-4-1 • 9-5-1 • 10-1-2, 10-1-3, 10-3-1, 10-3-2, 10-3-3 • 15-0-3, 15-1-3, 15-3-3, 15-3-5, 15-3-6, 15-3-7, 15-4-1, 15-5-1, 15-5-2 • 18-4-1	0

SQO Level 2

In addition to all the requirements of SQO Level 1, this level includes the following thresholds:

Metric	Threshold Value
Number of unjustified red checks	0
Number of unjustified `Non-terminating call` and `Non-terminating loop` checks	0

SQO Level 3

In addition to all the requirements of SQO Level 2, this level includes the following thresholds:

Metric	Threshold Value
Number of unjustified gray `Unreachable code` checks	0

SQO Level 4

In addition to all the requirements of SQO Level 3, this level includes the following thresholds:

Metric	Threshold Value
Percentage of justified orange checks, calculated as the number of green and justified orange checks divided by the total number of green and orange checks.	`Invalid C++ specific operations`: 50
	`Correctness condition`: 60
	`Division by zero`: 80
	`Uncaught exception`: 50
	`Function not returning value`: 80
	`Illegally dereferenced pointer`: 60
	`Return value not initialized`: 80
	`Non-initialized local variable`: 80
	`Non-initialized pointer`: 60
	`Non-initialized variable`: 60
	`Null this-pointer calling method`: 50
	`Incorrect object oriented programming`: 50
	`Out of bounds array index`: 80
	`Overflow`: 60
	`Invalid shift operations`: 80
	`User assertion`: 60

SQO Level 5

In addition to all the requirements of SQO Level 4, this level includes the following thresholds:

Metric	Threshold Value
Number of unjustified violations of the following MISRA C:2004 rules: • 6.3 • 8.7 • 9.2, 9.3 • 10.3, 10.5 • 11.1, 11.5 • 12.1, 12.2, 12.5, 12.6, 12.9, 12.10 • 13.1, 13.2, 13.6 • 14.8, 14.10 • 15.3 • 16.3, 16.8, 16.9 • 19.4, 19.9, 19.10, 19.11, 19.12 • 20.3	0
Number of unjustified violations of the following MISRA C:2012 rules: • 11.8 • 12.1 and 12.3 • 13.2 and 13.4 • 14.4 • 15.6 and 15.7 • 16.4 and 16.5 • 17.4 • 20.4, 20.6, 20.7, 20.9, and 20.11	0

Metric	Threshold Value
Number of unjustified violations of the following MISRA C++ rules: • 3-4-1, 3-9-2 • 4-5-1 • 5-0-1, 5-0-2, 5-0-7, 5-0-8, 5-0-9, 5-0-10, 5-0-13, 5-2-1, 5-2-2, 5-2-7, 5-2-11, 5-3-3, 5-2-5, 5-2-6, 5-3-2, 5-18-1 • 6-2-1, 6-3-1, 6-4-2, 6-4-6, 6-5-3 • 8-4-3, 8-4-4, 8-5-2, 8-5-3 • 11-0-1 • 12-1-1, 12-8-2 • 16-0-5, 16-0-6, 16-0-7, 16-2-2, 16-3-1	0
Percentage of justified orange checks, calculated as the number of green and justified orange checks divided by the total number of green and orange checks.	`Invalid C++ specific operations`: 70
	`Correctness condition`: 80
	`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

Metric	Threshold Value
	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.

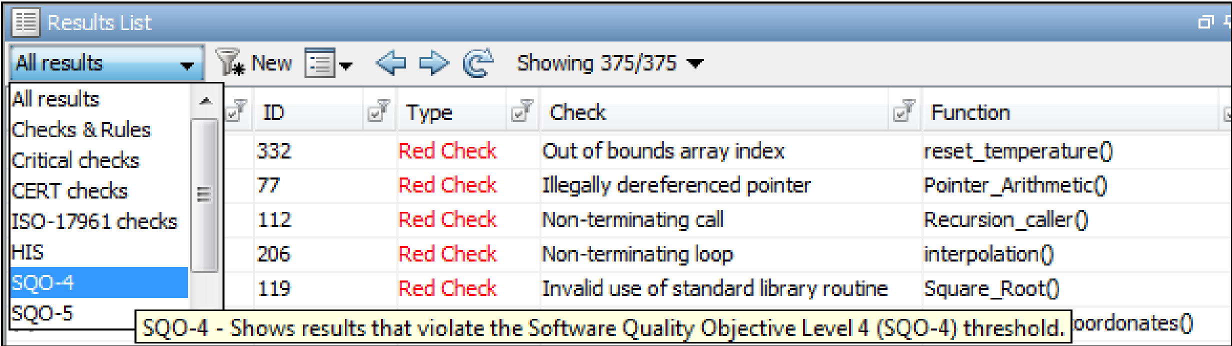
Metric	Threshold Value
Number of unjustified MISRA C and MISRA C++ coding rule violations	0
Number of unjustified red checks	0
Number of unjustified `Non-terminating call` and `Non-terminating loop` checks	0
Number of unjustified gray `Unreachable code` checks	0
Percentage of justified orange checks, 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.



For more information on:

- Comparing orange checks against SQOs, see "Limit Display of Orange Checks" on page 10-10.
- Comparing coding rule violations against SQOs, see "Suppress Certain Rules from Display in One Click" on page 12-15.
- Comparing code metrics against SQOs, see "Review Code Metrics" on page 8-24.

- 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 13-23.

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.

Project and Results Folder Contents

When you run an analysis, 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, see “Specify Results Folder” on page 6-2.

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`

Related Examples

- "Specify Results Folder" on page 6-2
- "Open Results" on page 6-5

Generate Report

This example shows how to generate a report from your verification results. Using a customizable template, the report presents your results in a concise manner for managerial review or other purposes. To generate a verification report, do one of the following:

- Specify certain options before verification so that the software automatically generates a report.
- Generate a report from your verification results.

You can also export your results to a text file and generate graphs and statistics. See “Export Polyspace Analysis Results” on page 8-136.

Specify Report Generation Before Verification

User Interface	Command Line
1. Select your project configuration. On the **Configuration** pane, select **Reporting**. Specify report generation options. For more information, see “Reporting”. 2. Run verification and open your results. 3. Select **Reporting** > **Open Report** 4. Navigate to the `Polyspace-Doc` subfolder in your results folder. You can see the generated report in this subfolder. Click **OK** to open the report.	Use the appropriate option with the `polyspace-code-prover-nodesktop` command (or `polyspaceCodeProver` function in MATLAB). For more information on the options, see the section **Command-Line Information** in “Reporting”. Additionally, you can also specify a report name using the option `-report-output-name`.

Generate Report After Verification

<table>
<tr><th>User Interface</th><th>Command Line</th></tr>
<tr><td>

1. Open your verification results.
2. Select **Reporting** > **Run Report**.

   The Run Report dialog box opens.
3. Select the following options:

   - In the **Select Reports** section, select the report templates you want to use. For example, you can select **Developer** and **Quality**.

     For more information, see `Bug Finder and Code Prover report (-report-template)`.
   - Select an **Output folder** in which to save the reports.
   - Select the **Output format** for the reports.
   - 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**.

</td><td>

Use the appropriate option with the `polyspace-report-generator` command.

The available options are:

- `-template` *`path`*: Path to report template file. For more information, see `Bug Finder and Code Prover report (-report-template)`.

  The predefined report templates are in *`matlabroot`*`\toolbox\polyspace\psrptgen\templates\Developer.rpt`. Here, *`matlabroot`* is the MATLAB installation folder such as `C:\Program Files\MATLAB\R2015a`.
- `-format` *`type`*: Output format of report. The allowed *`type`*s are`HTML`, `PDF` and `WORD`.
- `-output-name` *`filename`*: Name of report.
- `-results-dir` *`folder_paths`*: Path to folder containing your verification results.

  To generate a single report for multiple verifications, specify *`folder_paths`* as follows:

  ```
  "folder1, folder2, ..., folderN"
  ```

  where *`folder1, folder2, ...`* are paths to the folders that contain verification results. For example,

</td></tr>
</table>

User Interface	Command Line
For more information on filtering, see “Filter and Group Results” on page 8-113. • If you perform a file by file verification, you can generate a report of the verification results for each file or for all the files together. To generate a single report, select the option **Generate a single report including all unit results**. You can generate a single *filtered* report for all units. The report uses the filters applied to each unit and the review scope (`All results`, `Critical checks`, etc.) applied to the currently displayed unit. 4 Click **Run Report**. The software creates the specified reports and opens them.	`"C:\My_project \Module_1\results, C: \My_project\Module_2\Results"` If you do not specify a folder path, the software uses verification results from the current folder. • `-set-language-english`: Use this option to generate English reports if the default report is in another language. The display language (Windows) or locale (Linux) of your operating system determines the default language in the report.

See Also

Generate report | Bug Finder and Code Prover report (-report-template) | Output format (-report-output-format)

Related Examples

- “Customize Existing Report Template” on page 8-144

Export Polyspace Analysis Results

You can export your verification 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 check type (**Division by zero**, **Overflow**), you can calculate how many checks are red, orange, or green.
- Integrate the verification 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

1. Open your verification results.
2. Export the full set of results or only a subset of the results.
• To export all results, select **Reporting** > **Export** > **Export All Results**.
• If you want to filter results from your report, use filters on the **Results List** pane to display only the results that you want to report. Then, when exporting results, select **Reporting** > **Export** > **Export Currently Displayed Results**.
For more information on filtering, see “Filter and Group Results” on page 8-113.
3. Select a location to save the text file and click **OK**.
	Use appropriate options with the `polyspace-report-generator` command.
	The available options are:
	• `-generate-results-list-file`: Specifies that a text file must be generated.
	• `-results-dir` *`folder_paths`*: Path to folder containing your verification results. If you do not specify a folder path, the software uses verification results from the current folder.
	To generate text files for multiple verifications, specify *`folder_paths`* as follows:
	`"folder1, folder2, ..., folderN"`
	`folder1`, *`folder2`*, `...` are paths to the folders that contain verification results. For example:
	`"C:\My_project\Module_1\results, C:\My_project\Module_2\Results"`
	To merge the text files, use the `join` function.

The exported text file uses the character encoding on your operating system. If you encounter issues with special characters from your comments, 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 Polyspace Analysis Results in MATLAB” on page 8-140.

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). Some of the result information includes:

- ID: Unique number for a result for the current verification
- Family: Run-time Check, Global Variable, MISRA C:2012, and so on.
- Group: Check groups, MISRA C:2012 groups, etc.
- Check: Run-time check names, MISRA C:2012 coding rule number, and so on.
- Color: Run-time check colors on page 8-3
- Detail: Additional information about a result. This shows the first line of the **Result Details** pane.
- New: Whether the result is new compared to the last verification on the same code
- Full path to file
- Function
- Status, Severity, Comment: Information that *you* enter about a result.

For more information, see “Results List” on page 8-88. Though you cannot identify the location of a result in your source code using the text file, you can parse the file and generate graphs or statistics about your results.

The text file or the table also contains a **Key** column. The entry in this column is unique to a result across multiple verifications. When you merge multiple verification results that might contain common files, use this entry to eliminate copies of a result. For instance, if you run coding-rule checking on multiple modules and merge the results, header files and coding rule violations in them appear in multiple module verification results. To eliminate copies of a coding rule violation, use the entry in the **Key** column.

See Also

Related Examples

- "Visualize Polyspace Analysis Results in MATLAB" on page 8-140

Visualize Polyspace 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, if the folder `C:\MyResults` contains results of a Polyspace analysis, enter the following:

```
resObj = polyspace.CodeProverResults('C:\MyResults');
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 6-37.

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(matlabroot,'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 ' ...
```

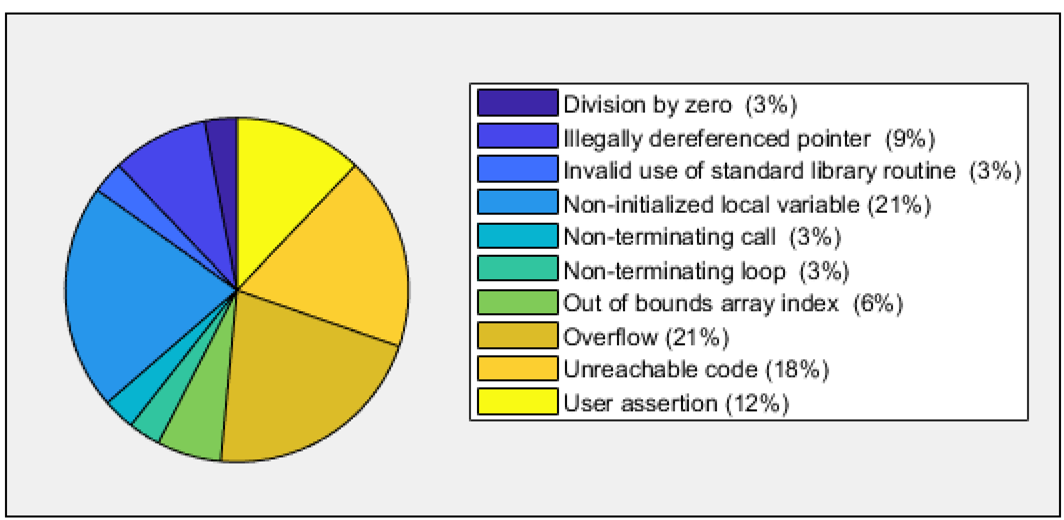
```
                 'in your code.'];
append(report, Paragraph(paragraphText));

% 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.
- `mlreportgen.dom.Document.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 8-136

Customize Existing 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 Report” on page 8-133.
- 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.

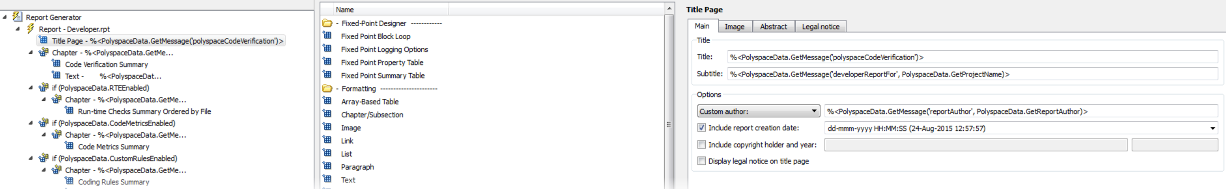
In this example, you view the components of the **Developer** template.

1. Open the Report Explorer interface of Simulink Report Generator. At the MATLAB command prompt, enter:

   ```
   report
   ```
2. Open the **Developer** template in the Report Explorer interface.

The **Developer** template is in *`matlabroot`*`/toolbox/polyspace/psrptgen/templates` where *`matlabroot`* is the MATLAB installation folder. Use the `matlabroot` command to find the installation folder location.

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:

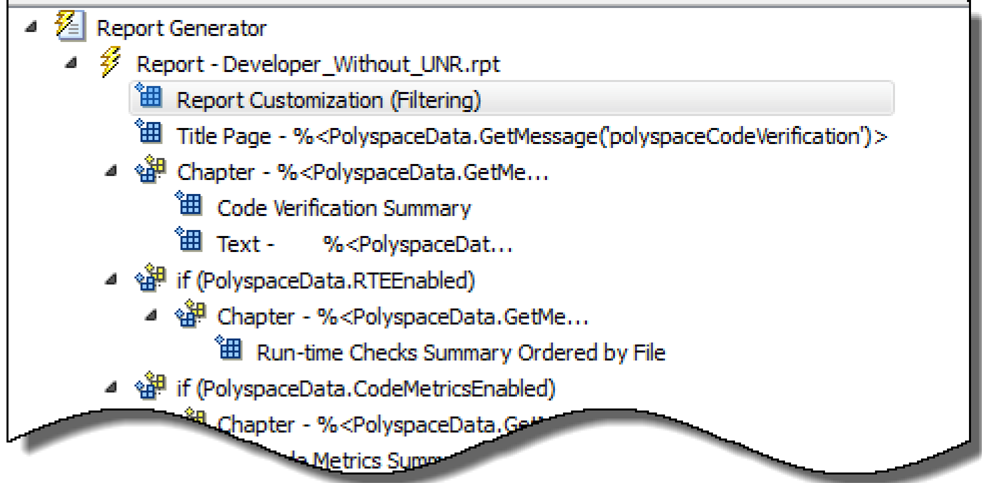
- Change properties of existing components of your template.
- Add new components to your template or remove existing components.

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.

Advanced Filters

Justification status: All

For each filter give a list of regular expressions matching the content to include in the report. Example to inlcude only Assert and Out of bounds checks in the report in the check types filter enter:
User assertion
Out of bounds array index

Files to include: .*

Check types to include: ^(?!Unreachable code).*

You can enter MATLAB regular expressions in this field using the Polyspace result names. 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**.

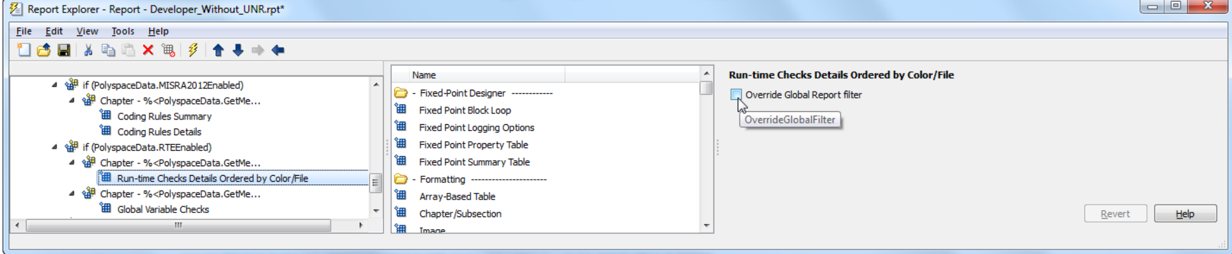
3 Change an existing chapter-specific component so that it does not override the global filter you applied in the previous step. If you prevent the overriding, the chapter-specific component follows the filtering specifications in the global component.

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.psprj**.

The demo result contains **Unreachable code** checks.

b Create a pdf report using the **Developer** template. See “Generate Report” on page 8-133.

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 8-150.

See Also

`Bug Finder and Code Prover report (-report-template)` | `Generate report` | `Output format (-report-output-format)`

Related Examples

- "Generate Report" on page 8-133
- "Sample Report Template Customizations" on page 8-150

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 Report Template” on page 8-144.

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 `*matlabroot*/toolbox/polyspace/psrptgen/templates`. *`matlabroot`* is the MATLAB 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`.

```
Report - CodeMetrics.rpt
    Title Page - %<PolyspaceData.GetMessage('polyspaceCodeVerification')>
    Chapter - %<PolyspaceData.GetMe...
        Code Metrics Summary
        Recursive Functions
        if (PolyspaceData.CodeMetricsEnabled)
            if (~PolyspaceData.ADAEnabled || (PolyspaceData.ADAEnabled &&
                Code Metrics Details
    Chapter - %<PolyspaceData.GetMe...
        Configuration Parameters
```

When you generate a report by using the modified template, you see a table with the list of recursive functions.

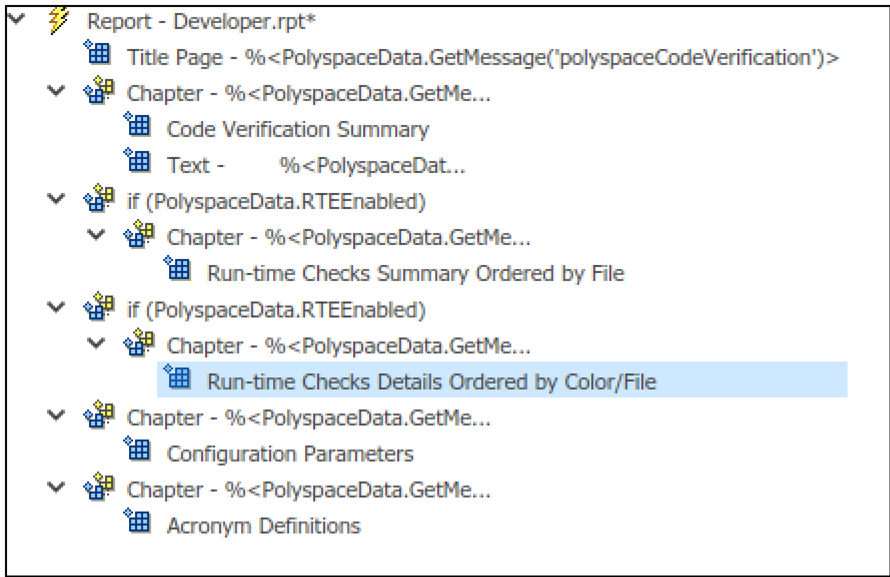
Show Red Run-Time Checks Only

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.

Title Page - %<PolyspaceData.GetMessage('polyspaceCodeVerification')>
Report Customization (Filtering)
Chapter - %<PolyspaceData.GetMe...
Code Verification Summary
Text - %<PolyspaceDat...
if (PolyspaceData.RTEEnabled)
Chapter - %<PolyspaceData.GetMe...
Run-time Checks Summary Ordered by File

Select the component. On the right pane, in the group **Advanced Filters**, from the **Justification Status** list, select `Un-justified`.

When you generate a report by using the modified template, you see only the non-justified run-time checks.

Add Chapter for Functional Design Errors

Suppose that you implement functional design testing using `assert` statements in your code. For instance, to test if the output of a function `out` is within a range `[MIN,MAX]`, your code uses the statement:

```
assert(MIN <= out && out <= MAX);
```

Polyspace runs the check `User assertion` to determine if the `assert` condition fails. Suppose that you want to report these checks in a separate chapter because they are different from the other run-time error checks.

Start from the **Developer** template. Make a copy of the chapter containing the component `Run-time Checks Details Ordered by Color/File`.

if (PolyspaceData.RTEEnabled)
Chapter - %<PolyspaceData.GetMe...
Run-time Checks Details Ordered by Color/File
Chapter - %<PolyspaceData.GetMe...
Run-time Checks Details Ordered by Color/File

Rename each of the two chapter titles so that you can distinguish between them. In each chapter, modify the component **Run-time Checks Details Ordered by Color/File** as follows:

- In one chapter, exclude **User assertion** checks. Select the component. On the right pane, in the group **Advanced Filters**, for **Check types to include**, enter:

  ```
  ^(?!User assertion).*
  ```

- In the other chapter, include **User assertion** checks. Select the component. On the right pane, in the group **Advanced Filters**, for **Check types to include**, enter:

  ```
  User assertion
  ```

  Clear the boxes for grey checks, because the **User assertion** checks cannot be grey.

When you generate a report by using the modified template, you see two copies of the chapter on run-time checks. The first chapter contains all checks other than **User assertion** checks and the second chapter contains **User assertion** checks only.

See Also

Related Examples

- “Customize Existing Report Template” on page 8-144

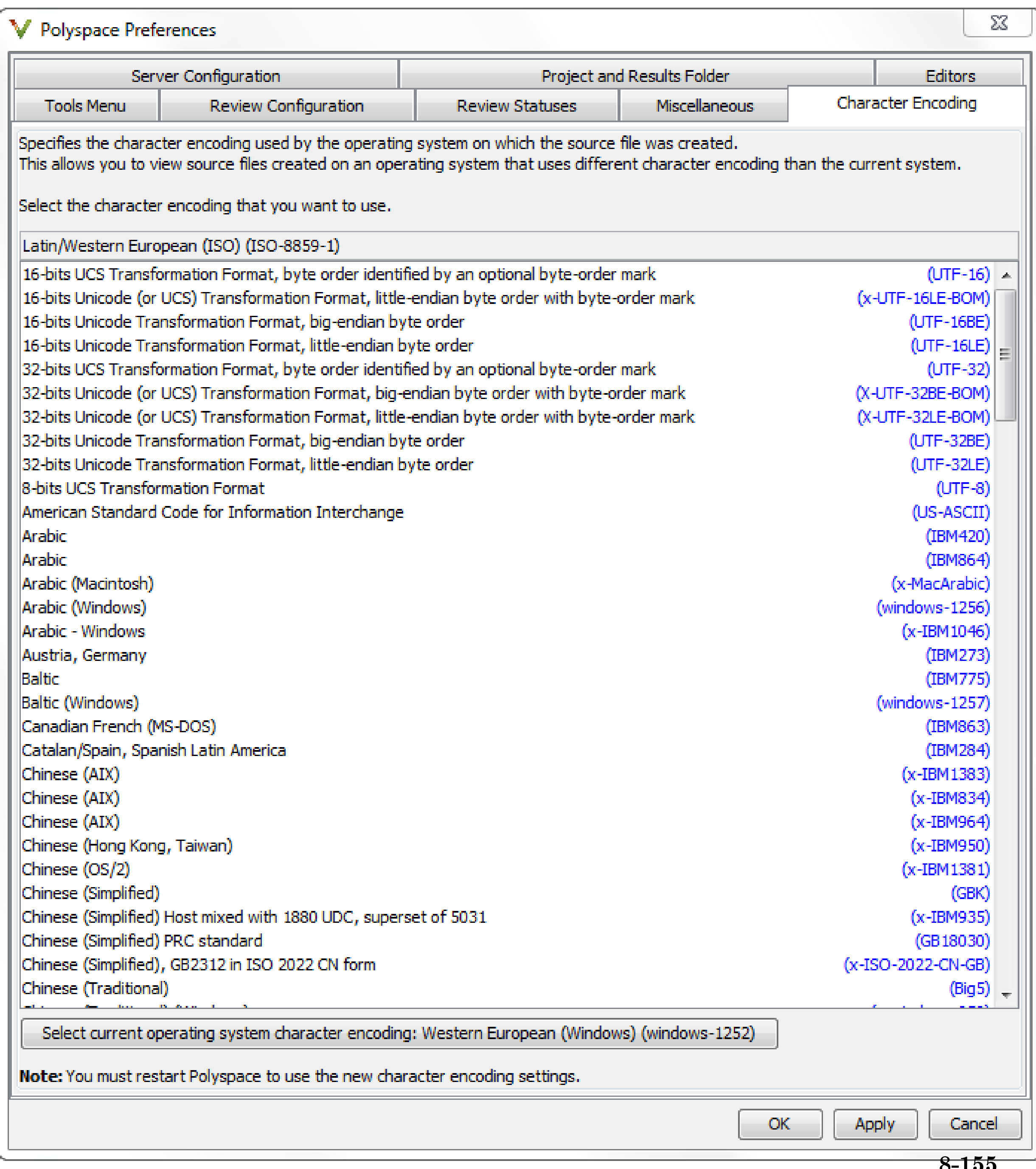
Set Character Encoding Preferences

If the source files that you want to verify are created on an operating system that uses different character encoding than your current system (for example, when viewing files containing Japanese characters), you receive an error message when you view the source file or run certain macros.

The **Character encoding** option allows you to view source files created on an operating system that uses different character encoding than your current system.

To set the character encoding for a source file:

1. Select **Tools** > **Preferences**.
2. In the Polyspace Preferences dialog box, select the **Character encoding** tab.



Polyspace Preferences
Server Configuration
Project and Results Folder
Editors
Tools Menu
Review Configuration
Review Statuses
Miscellaneous
Character Encoding
Specifies the character encoding used by the operating system on which the source file was created.
This allows you to view source files created on an operating system that uses different character encoding than the current system.
Select the character encoding that you want to use.
Latin/Western European (ISO) (ISO-8859-1)
16-bits UCS Transformation Format, byte order identified by an optional byte-order mark (UTF-16)
16-bits Unicode (or UCS) Transformation Format, little-endian byte order with byte-order mark (x-UTF-16LE-BOM)
16-bits Unicode Transformation Format, big-endian byte order (UTF-16BE)
16-bits Unicode Transformation Format, little-endian byte order (UTF-16LE)
32-bits UCS Transformation Format, byte order identified by an optional byte-order mark (UTF-32)
32-bits Unicode (or UCS) Transformation Format, big-endian byte order with byte-order mark (X-UTF-32BE-BOM)
32-bits Unicode (or UCS) Transformation Format, little-endian byte order with byte-order mark (X-UTF-32LE-BOM)
32-bits Unicode Transformation Format, big-endian byte order (UTF-32BE)
32-bits Unicode Transformation Format, little-endian byte order (UTF-32LE)
8-bits UCS Transformation Format (UTF-8)
American Standard Code for Information Interchange (US-ASCII)
Arabic (IBM420)
Arabic (IBM864)
Arabic (Macintosh) (x-MacArabic)
Arabic (Windows) (windows-1256)
Arabic - Windows (x-IBM1046)
Austria, Germany (IBM273)
Baltic (IBM775)
Baltic (Windows) (windows-1257)
Canadian French (MS-DOS) (IBM863)
Catalan/Spain, Spanish Latin America (IBM284)
Chinese (AIX) (x-IBM1383)
Chinese (AIX) (x-IBM834)
Chinese (AIX) (x-IBM964)
Chinese (Hong Kong, Taiwan) (x-IBM950)
Chinese (OS/2) (x-IBM1381)
Chinese (Simplified) (GBK)
Chinese (Simplified) Host mixed with 1880 UDC, superset of 5031 (x-IBM935)
Chinese (Simplified) PRC standard (GB18030)
Chinese (Simplified), GB2312 in ISO 2022 CN form (x-ISO-2022-CN-GB)
Chinese (Traditional) (Big5)
Select current operating system character encoding: Western European (Windows) (windows-1252)
Note: You must restart Polyspace to use the new character encoding settings.
OK
Apply
Cancel

3. Select the character encoding used by the operating system on which the source file was created.
4. Click **OK**.
5. Close and restart the Polyspace verification environment to use the new character encoding settings.

Reviewing Checks

- "Review and Fix Absolute Address Usage Checks" on page 9-3
- "Review and Fix Correctness Condition Checks" on page 9-4
- "Review and Fix Division by Zero Checks" on page 9-10
- "Review and Fix Function Not Called Checks" on page 9-16
- "Review and Fix Function Not Reachable Checks" on page 9-19
- "Review and Fix Function Not Returning Value Checks" on page 9-21
- "Review and Fix Illegally Dereferenced Pointer Checks" on page 9-23
- "Review and Fix Incorrect Object Oriented Programming Checks" on page 9-31
- "Review and Fix Invalid C++ Specific Operations Checks" on page 9-34
- "Review and Fix Invalid Shift Operations Checks" on page 9-37
- "Review and Fix Invalid Use of Standard Library Routine Checks" on page 9-43
- "Invalid Use of Standard Library Floating Point Routines" on page 9-46
- "Review and Fix Non-initialized Local Variable Checks" on page 9-50
- "Review and Fix Non-initialized Pointer Checks" on page 9-54
- "Review and Fix Non-initialized Variable Checks" on page 9-57
- "Review and Fix Non-Terminating Call Checks" on page 9-60
- "Identify Function Call with Run-Time Error" on page 9-63
- "Review and Fix Non-Terminating Loop Checks" on page 9-65
- "Identify Loop Operation with Run-Time Error" on page 9-69
- "Review and Fix Null This-pointer Calling Method Checks" on page 9-72
- "Review and Fix Out of Bounds Array Index Checks" on page 9-74
- "Review and Fix Overflow Checks" on page 9-79
- "Detect Overflows in Buffer Size Computation" on page 9-84
- "Review and Fix Return Value Not Initialized Checks" on page 9-86
- "Review and Fix Uncaught Exception Checks" on page 9-90
- "Review and Fix Unreachable Code Checks" on page 9-93

- "Review and Fix User Assertion Checks" on page 9-99
- "Find Relations Between Variables in Code" on page 9-104

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.

- To add a justification in your result, see “Add Review Comments to Results” on page 8-32.
- To add a justification in your code, see “Justify Results Through Code Annotations” on page 8-36.

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:

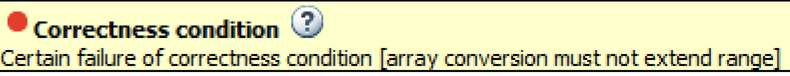
- “Review Red Checks” on page 8-10
- “Review Orange Checks” on page 8-17

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:

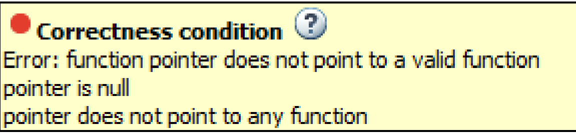
- An array is converted to another array of larger size.

  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.

Correctness condition
Warning: function pointer may not point to a valid function
Pointer is not null.
Polyspace assumption: Verification continues with full range for return values and modifiable arguments.
This check may be an issue related to unbounded input values
Absolute address assignment in function_pointer_uses_abs_addr.c line 3 may lead to imprecision here

- 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:

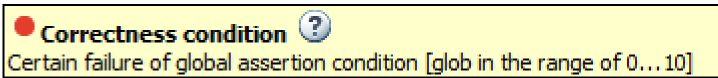
  - The function pointer points to a function `func`.
  - `func` returns an `int` value, but the return type of the function pointer is `double`.

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 5-40.

  In the following example, a red check occurs because:

  - You specify a range 0...10 for the variable `glob`.
  - The value of the variable falls outside this range.



Step 2: Determine Root Cause of Check

Based on the check information on the **Result Details** pane, perform further steps to determine the root cause. You can perform the following steps in the Polyspace user interface only.

Check Information	How to Determine Root Cause
An array is converted to another array of larger size.	1 To determine the array sizes, see the definition of each array variable. Right-click the variable and select **Go To Definition**. 2 If you dynamically allocate memory to an array, it is possible that their sizes are not available during definition. Browse through all instances of the array variable to find where you allocate memory to the array. a Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. b On the **Search** pane, select the previous instances.

<table>
<tr><th>Check Information</th><th>How to Determine Root Cause</th></tr>
<tr><td>Issues when dereferencing a function pointer:
The function pointer has value <code>NULL</code> when dereferenced.
The function pointer does not point to a function when dereferenced.
The function pointer points to a function, but the argument types of the pointer and the function do not match.
The function pointer points to a function, but the argument numbers of the pointer and the function do not match.
The function pointer points to a function, but the return types of the pointer and the function do not match.</td><td>1 Find the location where you assign the function pointer to a function.
a Right-click the function pointer. Select Search For All References.
All instances of the function pointer appear on the Search pane with the current instance highlighted.
b On the Search pane, select the previous instances.
2 Determine the argument and return types of the function pointer type and the function. Identify if there is a mismatch between the two. For instance, in the following example, determine the argument and return types of <code>typeFuncPtr</code> and <code>func</code>.
<code>typeFuncPtr funcPtr = func;</code>
a Right-click the function pointer type and select Go To Definition.
b Right-click the function and select Go To Definition. If the definition does not exist, this option shows the function stub definition instead. In this case, find the function declaration.
3 Sometimes, you assign a function pointer to a function with matching signature, but the assignment is unreachable. Check if this is the case.</td></tr>
</table>

Check Information	How to Determine Root Cause
The value of a variable falls outside the range that you specify through the **Global Assert** mode.	Browse through all previous instances of the global variable. Identify a suitable point to constrain the variable. 1 Right-click the variable. Select **Show In Variable Access View**. 2 On the **Variable Access** pane, select each instance of the variable.

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 “Add Review Comments to Results” on page 8-32 and “Justify Results Through Code Annotations” on page 8-36.

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:

- “Review Red Checks” on page 8-10
- “Review Orange Checks” on page 8-17

Step 1: Interpret Check Information

Place your cursor on the `/` or `%` operation that causes the **Division by zero** error.

```
func(1.0/val);
```

Probable cause for 'Division by Zero': Stubbed function 'getVal'
operator / on type float 64
left: 1.0
right: $[-2.1475E^{+09} .. 2.1475E^{+09}]$
result: $[-1.0001 .. -4.6566E^{-10}]$ or $[4.6566E^{-10} .. 1.0001]$

Obtain the following information from the tooltip:

- The values of the right operand (denominator).

  In the preceding example, the right operand, `val`, has a range that contains zero.

  Possible fix: To avoid the division by zero, perform the division only if `val` is not zero.

Integer	Floating-point

`if(val != 0)`
`    func(1.0/val);`
`else`
`    /* Error handling */`
	`#define eps 0.0000001`
	`.`
	`.`
	`if(val < -eps || val > eps)`
	`    func(1.0/val);`
	`else`
	`    /* Error handling */`

- The probable root cause for division by zero, if indicated in the tooltip.

  In the preceding example, the software identifies a stubbed function, `getVal`, as probable cause.

  Possible fix: To avoid the division by zero, constrain the return value of `getVal`. For instance, specify that `getVal` returns values in a certain range, for example, `1..10`. For more information, see "Constrain Stubbed Functions" on page 5-44.

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 "Constrain Stubbed Functions" on page 5-44.

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	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 Double-click the variable on the **Source** pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with and read operations with .
Function return value `ret=func();`	1 Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration. 2 In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 9-104.

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 5-35.
- 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 “Add Review Comments to Results” on page 8-32 and “Justify Results Through Code Annotations” on page 8-36.

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. To:

- Add justification in your result, see "Justify Results Through Code Annotations" on page 8-36.
- Add justification in your code, see "Justify Results Through Code Annotations" on page 8-36.

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.

```
131
132  static s32 unused_fonction(void)
133  {
134      return saved_values[output_v1] + (s32) generic_validation((output_v6 / 10000) , (output_v7 / 16000));
135  }
```

Step 2: Determine Root Cause of Check

1 Search for the function name and see if you can find a call to the function in your code.

On the **Search** pane, enter the function name. From the drop-down list beside the search field, select **Source**.

Possible fix: If you do not find a call to the function, determine why the function definition exists in your code.

2. If you find a call to the function, see if it occurs in the body of another uncalled function.

   Possible fix: Investigate why the latter function is not called.

3. See if you call the function indirectly, for example, through function pointers.

   If the indirection is too deep, Polyspace sometimes cannot determine that a certain function is called.

   Possible fix: If Polyspace cannot determine that you are calling a function indirectly, you must verify the function separately. You do not need to write a new `main` function for this other verification. Polyspace can generate a `main` function if you do not provide one in your source. You can change the `main` generation options if needed. For more information on the options, see "Code Prover Verification".

Step 3: Look for Common Causes of Check

Look for the following common causes of the **Function not called** check.

- Determine if you intended to call the function but used another function instead.
- Determine if you intended to replace some code with a function call. You wrote the function definition, but forgot to replace the original code with the function call.

  If this situation occurs, you are likely to have duplicate code.
- See if you intend to call the function from yet unwritten code. If so, retain the function definition.
- For code intended for multitasking, see if you have specified all your entry point functions.

  To see the options used for the result, select the link **View configuration for results** on the **Dashboard** pane.

  For more information, see `Entry points (-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 "Manually Model Tasks if main Contains Infinite Loop" on page 5-72.

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. To:

- Add justification in your result, see “Add Review Comments to Results” on page 8-32.
- Add justification in your code, see “Justify Results Through Code Annotations” on page 8-36.

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.

```
void increase(int* arr, int index);

void printError()
{
  printf("Array index exceeds array size.");
}
```

Step 2: Determine Root Cause of Check

Determine where the function is called and review why all the function call sites are unreachable. You can perform the following steps in the Polyspace user interface only.

1 Select the check on the **Results List** pane.
2 On the **Result Details** pane, click the *fx* button.

   On the **Call Hierarchy** pane, you see the callers of the function denoted by ◂ .
3 On the **Call Hierarchy** pane, select each caller.

   This action takes you to the function call on the **Source** pane.
4 See if the caller itself is called from unreachable code. If the caller definition is entirely in gray on the **Source** pane, it is called from unreachable code. Follow the same investigation process, starting from step 1, for the caller.
5 Otherwise, investigate why the section of code from which you call the function is unreachable.

   The code can be unreachable because it follows a red check or because it contains the gray **Unreachable code** check.

   - If a red check occurs, fix your code to remove the check. See "Review Red Checks" on page 8-10.
   - 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 9-93.

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

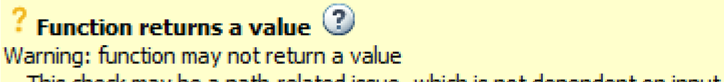
Follow one or more of these steps until you determine a fix for the **Function not returning value** check. For a description of the check and code examples, see `Function not returning value`.

For the general workflow that applies to all checks, see:

- “Review Red Checks” on page 8-10
- “Review Orange Checks” on page 8-17

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:

- “Review Red Checks” on page 8-10
- “Review Orange Checks” on page 8-17

Step 1: Interpret Check Information

Place your cursor on the dereference operator.

Obtain the following information from the tooltip:

- Whether the pointer can be `NULL`.

  In the following example, `ptr` can be `NULL` when dereferenced.

```
*ptr = 1;
```

```
Probable cause for 'Illegally dereferenced pointer': Stubbed function 'input'
Dereference of local pointer 'ptr' (pointer to int 32, size: 32 bits):
    Pointer may be null.
    Points to 4 bytes at unknown offset in buffer of unknown size, so may be outside bounds.
    Pointer may point to dynamically allocated memory.
Assignment to dereference of local pointer 'ptr' (int 32): 1
```

  Possible fix: Dereference `ptr` only if it is not `NULL`.

```
if(ptr !=NULL)
    *ptr = 1;
```

```
else
    /* Alternate action */
```

- Whether 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 “Constrain Stubbed Functions” on page 5-44.

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.

<table>
<tr><th>Error</th><th>How to Find Root Cause</th></tr>
<tr><td>Pointer can be <code>NULL</code>.</td><td>Find an execution path where the pointer is assigned the value NULL or not assigned a definite address.

1 Right-click the pointer and select Search For All References.
2 Find each previous instance where the pointer is assigned an address.
3 For each instance, on the Source pane, place your cursor on the pointer. The tooltip indicates whether the pointer can be <code>NULL</code>.

Possible fix: If the pointer can be <code>NULL</code>, place a check for <code>NULL</code> immediately after the assignment.
<pre>if(ptr==NULL)
  /* Error handling*/
else {
  .
  .
    }</pre>4 If the pointer is not <code>NULL</code>, 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.</td></tr>
<tr><td>Pointer can point to dynamically allocated memory.</td><td>Identify where the allocation occurs.

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>.

Possible fix: After the allocation, test the pointer for <code>NULL</code>.</td></tr>
</table>

<table>
<tr><th>Error</th><th>How to Find Root Cause</th></tr>
<tr><td>Pointer can point outside bounds allowed by the buffer.</td><td>1 Find the allowed buffer.
a On the Search tab, enter the name of the variable that the pointer points to. You already have this name from the tooltip on the check.
b Search for the variable definition. Typically, this is the first search result.

If the variable is an array, note the array size. If the variable is a structure, search for the structure type name on the Search tab and find the structure definition. Note the size of the structure field that the pointer points to.
2 Find out why the pointer points outside the allowed buffer.
a Right-click the pointer and select Search For All References.
b Identify any increment or decrement of the pointer. See if you intended to make the increment or decrement.

Possible fix: Remove unintended pointer arithmetic. To avoid pointer arithmetic that takes a pointer outside 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.</td></tr>
</table>

Step 3: Look for Common Causes of Check

Look for common causes of the **Illegally dereferenced pointer** check.

- If you use pointers for moving through an array, see if you can use an array index instead.

  To avoid use of pointer arithmetic in your code, look for violations of MISRA C: 2004 rule 17.4 or MISRA C: 2012 rule 18.4. For more information, see “Select Specific MISRA or JSF Coding Rules” on page 12-3.

- 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 9-63.
- 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 “Add Review Comments to Results” on page 8-32 and “Justify Results Through Code Annotations” on page 8-36.

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

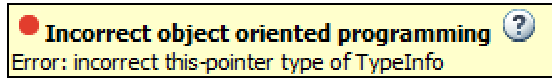
In this section...
“Step 1: Interpret Check Information” on page 9-31
“Step 2: Determine Root Cause of Check” on page 9-32

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 “Review Red Checks” on page 8-10.

Step 1: Interpret Check Information

On the **Results List** pane, select the check. The **Result Details** pane displays further information about the check.



You can see:

- The immediate cause of the check. For instance:
  - You dereference a function pointer that has the value `NULL` or points to an invalid member function.

    The member function is invalid if its argument or return type does not match the pointer argument or return type.
  - You call a pure `virtual` member function of a class from the class constructor or destructor.
  - You call a member function using an incorrect `this` pointer.

    To see why the `this` pointer can be incorrect, see `Incorrect object oriented programming`.
- The probable root cause of the check, if indicated.

Step 2: Determine Root Cause of Check

If you cannot determine the root cause based on the check information, use navigation shortcuts in the user interface to navigate to the root cause.

Based on the specific error, use one of the following methods to find the root cause.

Error	How to Find Root Cause
You dereference a function pointer that has the value `NULL`.	Right-click the function pointer and select **Search For All References**. Find the instance where you assign `NULL` to the function pointer.
You dereference a function pointer that points to an invalid member function.	Compare the argument and return types of the function pointer and the member function that it points to. 1 Right-click the function pointer on the **Source** pane and select **Search For All References**. Find the instances where you: • Define the function pointer. • Assign the address of a member function to the function pointer. 2 Find the member function definition. Right-click the member function name on the **Source** pane and select **Go To Definition**.
You call a pure `virtual` member function from a constructor or destructor.	Find the member function declaration and determine whether you intended to declare it as `virtual` or pure `virtual`. Alternatively, determine if you can replace the call to the pure `virtual` function with another operation, for instance, a call to a different member function. 1 Right-click the function name on the **Source** pane and select **Search for *`function_name`* in All Source Files**. 2 Find the function declaration from the search results. A pure `virtual` function has a declaration such as: `virtual void func() = 0;`

<table>
<tr><th>Error</th><th>How to Find Root Cause</th></tr>
<tr><td>You call a member function using an incorrect <code>this</code> pointer.</td><td>Determine why the <code>this</code> pointer is incorrect.

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>.
Right-click the function pointer on the Source pane and select Search For All References.
Browse through all write operations on the pointer. Look for the following issues:
Cast between pointers of unrelated types.
Pointer arithmetic that takes a pointer outside its allowed buffer, for instance, the bounds of an array.
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.</td></tr>
</table>

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:

- “Review Red Checks” on page 8-10
- “Review Orange Checks” on page 8-17

Step 1: Interpret Check Information

On the **Results List** pane, select the check. The **Result Details** pane displays further information about the check.

Warning: typeid argument may be incorrect
This check may be an issue related to unbounded input values
If appropriate, applying DRS to stubbed function getShapePtr() in file_typeid.cpp line 53 may remove this orange.

You can see:

- The immediate cause of the check. For instance:
  - The size of an array is not strictly positive.

    For instance, you create an array using the statement `arr = new char [num]`. `num` is possibly zero or negative.

    Possible fix: Use `num` as an array size only if it is positive.
  - The `typeid` operator dereferences a possibly `NULL` pointer.

    Possible fix: Before using the `typeid` operator on a pointer, test the pointer for `NULL`.

  - The `dynamic_cast` operator performs an invalid cast.

    Possible fix: The invalid cast results in a `NULL` return value for pointers and the `std::bad_cast` exception for references. Try to avoid the invalid cast. Otherwise, if the invalid cast is on pointers, make sure that you test the return value of `dynamic_cast` for `NULL` before dereference. If the invalid cast is on references, make sure that you catch the `std::bad_cast` exception in a `try-catch` statement.

- The probable root cause of the check, if indicated.

Step 2: Determine Root Cause of Check

If you cannot determine the root cause based on the check information, use navigation shortcuts in the user interface to navigate to the root cause.

Based on the nature of the error, use one of the following methods to find the root cause.

Error	How to Find Root Cause
An array size is nonpositive.	**1** Trace the data flow for the size variable. Follow the same root cause investigation steps as for a **Division by Zero** check. See “Review and Fix Division by Zero Checks” on page 9-10. **2** Identify a point where you can constrain the array size variable to positive values.
The `typeid` operator dereferences a possibly `NULL` pointer.	**1** Trace the data flow for the pointer variable. Follow the same root cause investigation steps as for an **Illegally dereferenced pointer** check. See “Review and Fix Illegally Dereferenced Pointer Checks” on page 9-23. **2** Identify a point where you can test the pointer for `NULL`.
The `dynamic_cast` operator performs an invalid cast.	Navigate to the definitions of the classes involved. Determine the inheritance relationship between the classes. **1** On the **Source** pane in the Polyspace user interface, right-click the class name. **2** Select **Go To Definition**.

Step 3: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Add Review Comments to Results” on page 8-32 and “Justify Results Through Code Annotations” on page 8-36.

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:

- "Review Red Checks" on page 8-10
- "Review Orange Checks" on page 8-17

Step 1: Interpret Check Information

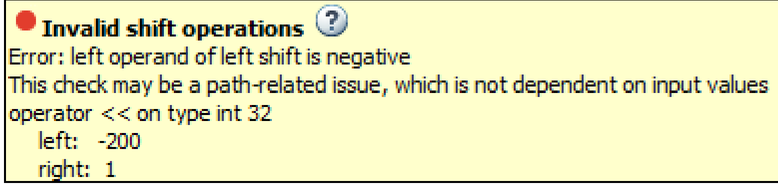
Select the red or orange **Invalid shift operations** check. Obtain the following information from the **Result Details** pane:

- 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.

```
? Invalid shift operations
Warning: left operand of left shift may be negative
This check may be an issue related to unbounded input values
If appropriate, applying DRS to stubbed function getVal in shf_2.c line 9 may remove this orange.
If appropriate, applying DRS to stubbed function getVal in shf_2.c line 8 may remove this orange.
operator << on type int 32
   left:   [-65535 .. 65535]
   right:  10
   result: multiples of 1024 in [0 .. 67107840 (0x3FFFC00)]
```

In the preceding example, the software identifies a stubbed function, `getVal` as probable cause.

Possible fix: To avoid the orange check, constrain the return value of `getVal`. For instance, specify that `getVal` returns values in a certain range, for example, `0..10`. For more information, see "Constrain Stubbed Functions" on page 5-44.

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 “Constrain Stubbed Functions” on page 5-44.

- 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 “Constrain Stubbed Functions” on page 5-44.
  - If you want Polyspace to allow the operation, use the analysis option **Allow negative operand for left shifts**. See `Allow negative operand for left shifts (-allow-negative-operand-in-shift)`.

To trace the data flow, select the check and note the information on the **Result Details** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Result Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.
- Otherwise:

  1. Find the previous write operation on the variable you want to trace.
  2. At the previous write operation, identify a new variable to trace back.

     Place your cursor on the variables involved in the write operation to see their values. The values help you decide which variable to trace.

3 Find the previous write operation on the new variable. Continue tracing back in this way until you identify a point to specify your constraint.

Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 Double-click the variable on the **Source** pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with and read operations with .

<table>
<tr><th>Variable</th><th>How to Find Previous Instances of Variable</th></tr>
<tr><td>Function return value

<code>ret=func();</code></td><td>1 Find the function definition.

Right-click <code>func</code> on the Source pane. Select Go To Definition, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration.
2 In the definition of <code>func</code>, identify each <code>return</code> statement. The variable that the function returns is your new variable to trace back.</td></tr>
</table>

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 9-104.

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 “Add Review Comments to Results” on page 8-32 and “Justify Results Through Code Annotations” on page 8-36.

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`.

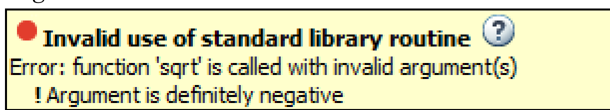
Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see:

- "Review Red Checks" on page 8-10
- "Review Orange Checks" on page 8-17

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** You can use the macro `EOF` as argument.

<table>
<tr><th>Function</th><th colspan="2">Cause of Red or Orange Check</th></tr>
<tr><td rowspan="9">Functions in math.h</td><td colspan="2">The software checks for multiple kinds of errors in sequence. The software performs each check only for those execution paths where the previous check passes.

Some examples are given below. For more information and a list of functions, see “Invalid Use of Standard Library Floating Point Routines” on page 9-46.</td></tr>
<tr><td>sqrt</td><td>The value of the argument can be negative.</td></tr>
<tr><td>pow</td><td>The first argument can be negative while the second argument is a non-integer.</td></tr>
<tr><td>exp, exp2, or the hyperbolic functions</td><td>The argument can be so large that the result exceeds the value allowed for a double.</td></tr>
<tr><td>log</td><td>The argument can be zero or negative.</td></tr>
<tr><td>asin or acos</td><td>The argument can be outside the range [-1,1].</td></tr>
<tr><td>tan</td><td>The argument can have the value HALF_PI.</td></tr>
<tr><td>acosh</td><td>The argument can be less than 1.</td></tr>
<tr><td>atanh</td><td>The argument can be greater than 1 or less than -1.</td></tr>
<tr><td>fprintf, fscanf, and other file handling functions</td><td colspan="2">The file pointer argument can be non-readable. For example, it can be NULL.</td></tr>
<tr><td>Functions that take string arguments</td><td colspan="2">The string argument can be an invalid string. For example, it does not end with a terminating '\0'.</td></tr>
</table>

Function	Cause of Red or Orange Check
`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 “Add Review Comments to Results” on page 8-32 and “Justify Results Through Code Annotations” on page 8-36.

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 “Constrain Stubbed Functions” on page 5-44. 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 `frexp` that take pointer arguments, the verification checks if it is valid to dereference the pointer. For instance, the pointer is not NULL or does not point outside allowed bounds.

The check looks for these errors in sequence.

- If the check finds a definite domain error, it does not look for the overflow error.
- If the check finds a possible domain error, it looks for the overflow error only for the execution paths where the domain error does not occur.

The check for each error itself can consist of multiple conditions, which are also checked in sequence. Each check is performed only for those execution paths where the previous check passes.

Single-Argument Functions Checked

The **Invalid Use of Standard Library Routine** check covers the following routines, their single-precision versions with suffix `f` (if they have one) and their long double versions with suffix `l`. The check works in exactly the same way for C and C++ code.

- `acos`
- `acosh`
- `asin`
- `asinh`
- `atan`
- `atanh`
- `ceil`
- `cos`
- `cosh`
- `exp`
- `exp2`
- `expm1`

- `fabs`
- `floor`
- `log`
- `log10`
- `log1p`
- `logb`
- `round`
- `sin`
- `sinh`
- `sqrt`
- `tan`
- `tanh`
- `trunc`
- `cbrt`

Functions with Multiple Arguments

The **Invalid Use of Standard Library Routine** check covers the following routines, their single-precision versions with suffix `f` (if they have one) and their long double versions with suffix `l`. The check works in exactly the same way for C and C++ code.

- `atan2`
- `fdim`
- `fma`
- `fmax`
- `fmin`
- `fmod`
- `frexp`
- `hypot`
- `ilogb`
- `ldexp`
- `modf`

- `nextafter`
- `nexttoward`
- `pow`
- `remainder`

See Also

`Consider non finite floats (-allow-non-finite-floats)` | `Float rounding mode (-float-rounding-mode)`

Review and Fix Non-initialized Local Variable Checks

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`.

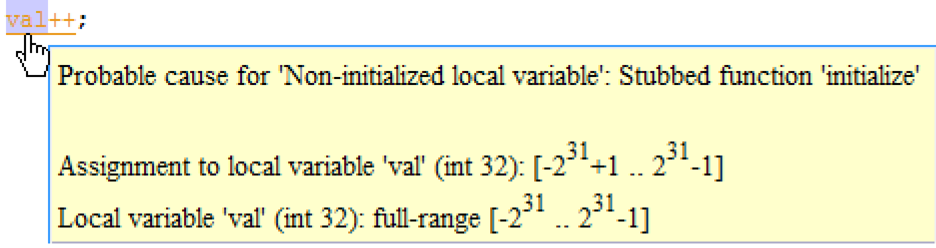
Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see:

- “Review Red Checks” on page 8-10
- “Review Orange Checks” on page 8-17

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 “Constrain Stubbed Functions” on page 5-44.

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 9-93.
- 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 “Add Review Comments to Results” on page 8-32 and “Justify Results Through Code Annotations” on page 8-36.

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:

- "Review Red Checks" on page 8-10
- "Review Orange Checks" on page 8-17

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 9-93.
- In a code block that is not reached on certain execution paths. For example, the operation occurs in an `if` block in a function. The `if` block is not entered for certain function inputs.

  Possible fix: Perform a write operation on all the execution paths. In the preceding example, perform the write operation in all branches of the `if ... elseif ... else` statement.

Depending on the nature of the variable, use the appropriate method to find previous operations on the variable. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Operations on Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 On the **Source** pane, double-click the variable. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.

Variable	How to Find Previous Operations on Variable
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. The current instance of the variable is shown on the **Variable Access** pane. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with ◀. Read operations are indicated with ▶.

Step 3: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See "Add Review Comments to Results" on page 8-32 and "Justify Results Through Code Annotations" on page 8-36.

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:

- “Review Red Checks” on page 8-10
- “Review Orange Checks” on page 8-17

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 9-93.
   - 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 “Add Review Comments to Results” on page 8-32 and “Justify Results Through Code Annotations” on page 8-36.

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 “Review Red Checks” on page 8-10.

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 9-63.

Possible fix: Investigate the cause of the orange check. Change your code or the function arguments to avoid the orange check.

Step 2: Look for Common Causes of Check

To trace a **Non-terminating call** check on a function call to an orange check in the function body, try the following:

- If the function call contains arguments, in the function body, search for all instances of the function parameters. See if you can find an orange check related to the parameters. Because other calls to the same function do not cause an operation failure, it is likely that the failure is related to the function parameter values in the red call.

  In the following example, in the body of `func`, search for all instances of `arg1` and `arg2`. Right-click the variable name and select **Search For All References**.

  ```
  void func(int arg1, double arg2) {
    .
    .
  }

  void main() {
    int valInt1,valInt2;
    double valDouble1, valDouble2;
    .
    .
    func(valInt1, valDouble1);
    func(valInt2, valDouble2);
  }
  ```

  Because `valInt1` and `valDouble1` do not cause an operation failure in `func`, the failure might be due to `valInt2` or `valDouble2`. Because `valInt2` and `valDouble2` are copied to `arg1` and `arg2`, the orange check must occur in an operation related to `arg1` or `arg2`.

- If the function call does not contain arguments, identify what is different between various calls to the function. See if you can relate the source of this difference to an orange check in the function body.

For instance, if the function reads a global variable, different calls to the function can operate on different values of the global variable. Determine if the function body contains an orange check related to the global variable.

Identify Function Call with Run-Time Error

This tutorial shows how to identify the function call that causes a run-time error in the function body.

If a function contains two different colors on the same operation for two different calls, the software combines the call contexts and displays an orange check on the operation. For example, when some function calls cause a red or orange check on an operation in the function body and other calls cause a green check, in your verification results, the operation is orange.

You have to distinguish orange checks that arise from combination of call contexts because an orange check can arise from other causes. Using the option Sensitivity context, make this distinction by storing individual call contexts for a function.

In this tutorial, a function is called twice. You identify which function call causes a run-time error in the function body.

1 Run analysis on this code and open the results.

```
void func(int arg) {
    int loc_var = 0;
    loc_var = 1/arg;
}

void main(void) {
    int num = 1;
    func(num + 10);
    func(num - 1);
}
```

   A red **Non-terminating call** check appears on the second call to `func`. In the body of `func`, there is an orange **Division by zero** check on the `/` operation.

2 Specify that you want to store individual call context information for the function `func`.

   a In your project configuration, select the **Precision** node.

   b Select `custom` for **Sensitivity context**.

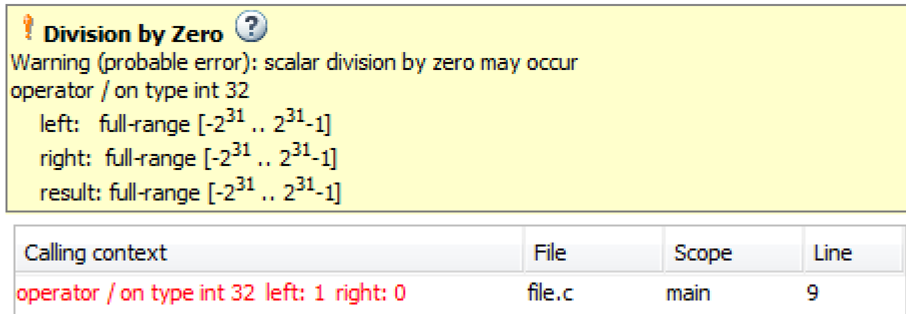
   c Click ➕ to add a new field. Enter `func`.

3 Run verification and open the results.

An orange **Division by zero** check still appears in the body of `func`. However, this orange check is marked on the **Results List** pane as a dark orange check and is denoted by a mark. Instead of a red **Non-terminating call** check, a dashed, red line appears on the second call to `func`.

4 Select the orange check.

The **Result Details** pane shows the call contexts for the check. You can see that one call produces a red check on the / operation and the other call produces a green check. You can click each call to navigate to it in your source code.



See Also

`Non-terminating call`

Related Examples

- “Review and Fix Non-Terminating Call Checks” on page 9-60
- “Test Orange Checks for Run-Time Errors” on page 10-19

More About

- “Sources of Orange Checks” on page 10-2

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 “Review Red Checks” on page 8-10.

Step 1: Interpret Check Information

Place your cursor on the loop keyword such as `for` or `while`.

Obtain the following information from the tooltip:

- Whether the loop is infinite or contains a run-time error.

  In the following example, it is likely that the loop is infinite.

  ```
  while(i<10) {
  ```
  loop may be infinite

- If the loop contains a run-time error, the number of loop iterations. Because Polyspace considers that execution stops when a run-time error occurs, from this number, you can determine which loop iteration contains the error.

  In the following example, it is likely that the loop contains a run-time error. The error is likely to occur on the 31st loop iteration.

  ```
  while(i<1024) {
  ```
  loop may fail due to a run-time error (maximum number of iterations: 31)

Step 2: Determine Root Cause of Check

- If the loop is infinite, determine why the loop-termination condition is never satisfied.

  If you deliberately have an infinite loop in your code, such as for cyclic applications, you can add a comment and justification in your result or code.

  - To add a justification in your result, see “Add Review Comments to Results” on page 8-32.

  - To add a justification in your code, see “Justify Results Through Code Annotations” on page 8-36.
- 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 9-69.

Step 3: Look for Common Causes of Check

- If the loop is infinite:
  - Check your loop-termination condition.
  - Inside the loop body, see if you change at least one of the variables involved in the loop-termination condition.

    For instance, if the loop-termination condition is `while (count1 + count2 < count3)`, see if you are changing at least one of `count1`, `count2`, or `count3` in the loop.
  - If you are changing the variables involved in the loop-termination condition, see if you are changing them in the right direction.

For instance, if the loop termination condition is `while(i<10)` and you decrement `i` in the loop, the loop does not terminate. You must increment `i`.

- If the loop contains a run-time error:
  - If the loop control variable is an array index, see if you have an orange **Out of bounds array index** error in the loop body.
  - If the loop control variable is passed to a function, see if you can relate the red **Non-terminating loop** error to an orange error in the function body.

Identify Loop Operation with Run-Time Error

This tutorial shows how to interpret Polyspace Code Prover results that highlight a run-time error inside a loop.

If an error occurs in a loop, the error shows in the analysis results as a red **Non-terminating loop** check on the loop keyword (`for`, `while`, and so on).

```
for (i = 0; i <= 10; i++)
```

The operation causing the error shows as an orange check in the loop. To distinguish this orange check from other orange checks in the loop, navigate directly from the red loop keyword to the operation responsible for the run-time error.

In this tutorial, a function is called in a loop. The function body contains another loop, which has an operation causing a run-time error. You trace from the first loop to the operation causing the run-time error.

1 Run verification on this code and open the results:

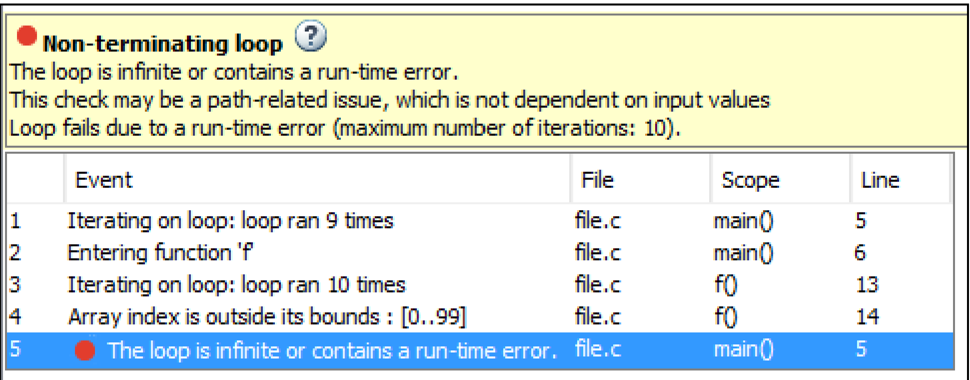
```
int a[100];

int f (int i);

void main() {
  int i,x=0;
  for (i = 0; i <= 10; i++) {
    x += f(i);
  }
}

int f (int i) {
  int j, x;
  x = 0;
  for (j = 0; j <= 10; j++) {
    x += a[10 + (i * j)];
  }
  return x;
}
```

2 Select the red **Non-terminating loop** result. The **Source** pane highlights the `for` loop in `main`.

3 Trace from the `for` loop in `main` to the operation causing the error. The operation is `x+= a[10 + (i*j)]`. An **Out of bounds array index** error occurs when `i` is 9 and `j` is 10. The error shows in orange on the `[` operator.

To trace from the red `for` keyword to the orange array access operation:

- Navigate directly to the operation. Right-click the `for` keyword and select **Go to Cause**.
- See the full history from the `for` keyword to the array access operation. Select the red `for` keyword. The **Result Details** pane shows the history.



You can read the event history in sequence. The outer loop loop runs nine times without error. On the tenth iteration (`i=9`), the loop enters the function `f`. Inside `f`, the inner loop runs ten times without error. On the eleventh iteration (`j=10`), the array access causes an error.

You can use this information to determine how to fix the run-time error on the array access operation.

Note You can navigate directly to the root cause of an error for loops with only a small number of iterations.

See Also

`Non-terminating loop`

Related Examples

- "Review and Fix Non-Terminating Loop Checks" on page 9-65
- "Test Orange Checks for Run-Time Errors" on page 10-19

More About

- "Sources of Orange Checks" on page 10-2

Review and Fix Null This-pointer Calling Method Checks

In this section...
"Step 1: Interpret Check Information" on page 9-72
"Step 2: Determine Root Cause of Check" on page 9-73

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:

- "Review Red Checks" on page 8-10
- "Review Orange Checks" on page 8-17

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:

- "Review Red Checks" on page 8-10
- "Review Orange Checks" on page 8-17

Step 1: Interpret Check Information

Place your cursor on the `[` symbol.

```
val = arr[i];
```

array size: 10
array index value: [0 .. 10]
Element of global array (int 32): full-range [-2^{31} .. $2^{31}-1$]

Obtain the following information from the tooltip:

- Array size. The allowed range for array index is 0 to (array size - 1).
- Actual range for array index

In the preceding example, the array size is 10. Therefore, the allowed range for the array index is 0 to 9. However, the actual range is 0 to 10.

Possible fix: To avoid the out of bounds array index, access the array only if the index is between 0 and (array size - 1).

```
#define SIZE 100
int arr[SIZE];
.
```

```
.
if(i<SIZE)
    val = arr[i]
else
    /*Error handling */
```

Step 2: Determine Root Cause of Check

If you want to know or change the array size, right-click the array variable and select **Go To Definition**, if the option exists. Otherwise, trace the data flow starting from the array index variable. Identify a point where you can constrain the index variable.

To trace the data flow, select the check, and note the information on the **Result Details** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Result Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.
- Otherwise:

  1. Find previous instances of the array index variable.
  2. Browse through those instances. Find the instance where you constrain the array index variable to (array size - 1).

     Possible fix: If you do not find an instance where you constrain the index variable, specify a constraint for the variable. For example:

     ```
     if(index<SIZE)
         read(array[index]);
     ```
  3. Determine if the constraint applies to the instance where the **Out of bounds array index** error occurs.

     For example, you can constrain the index variable in a `for` loop using `for(index=0; index<SIZE; index++)`. However, you access the array outside the loop where the constraint does not apply.

     Possible fix: Investigate why the constraint does not apply. See if you have to constrain the index variable again.

4. If the index variable is obtained from another variable, trace the data flow for the second variable. Determine if you have constrained that second variable to (array size - 1).

Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 Double-click the variable on the **Source** pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with and read operations with .

Variable	How to Find Previous Instances of Variable
Function return value `ret=func();`	**1** Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration. **2** In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 9-104.

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 “Add Review Comments to Results” on page 8-32 and “Justify Results Through Code Annotations” on page 8-36.

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:

- "Review Red Checks" on page 8-10
- "Review Orange Checks" on page 8-17

Step 1: Interpret Check Information

Place your cursor on the operation that overflows.

```
return(val*2);
```

Probable cause for 'Overflow': Stubbed function 'getVal'

operator * on type int 32

left: full-range [-2^{31} .. 2^{31}-1]
right: 2
result: even values in [-2^{31} .. 2147483646 (0x7FFFFFFE)]
(result is truncated)

Obtain the following information from the tooltip:

- The operand variable you can constrain to avoid the overflow.

  In the preceding example, the left operand, `val`, has full range of values.

  Possible fix: To avoid the overflow, perform the multiplication only if `val` is in a certain range.

```
if(val < INT_MAX/2)
    return(val*2);
else
    /* Alternate action */
```

- The probable root cause for overflow, if indicated in the tooltip.

  In the preceding example, the software identifies a stubbed function, `getVal`, as probable cause.

  Possible fix: To avoid the overflow, constrain the return value of `getVal`. For instance, specify that `getVal` returns values in a certain range, for example, `1..10`. For more information, see “Constrain Stubbed Functions” on page 5-44.

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 “Constrain Stubbed Functions” on page 5-44.

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.

<table>
<tr><th>Variable</th><th>How to Find Previous Instances of Variable</th></tr>
<tr><td>Local Variable</td><td>Use one of the following methods:
Search for the variable.
Right-click the variable. Select Search For All References.

All instances of the variable appear on the Search pane with the current instance highlighted.
On the Search pane, select the previous instances.
Browse the source code.
Double-click the variable on the Source pane.

All instances of the variable are highlighted.
Scroll up to find the previous instances.</td></tr>
<tr><td>Global Variable

Right-click the variable. If the option Show In Variable Access View appears, the variable is a global variable.</td><td>Select the option Show In Variable Access View.

On the Variable Access pane, the current instance of the variable is shown.
On this pane, select the previous instances of the variable.

Write operations on the variable are indicated with ◀ and read operations with ▶.</td></tr>
<tr><td>Function return value

<code>ret=func();</code></td><td>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.
In the definition of <code>func</code>, identify each <code>return</code> statement. The variable that the function returns is your new variable to trace back.</td></tr>
</table>

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 9-104.

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 “Add Review Comments to Results” on page 8-32 and “Justify Results Through Code Annotations” on page 8-36.

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 **Detect overflows** option, instead of the default value `signed`, use `signed-and-unsigned`. 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 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 **Detect overflows** drop-down list, select `signed`.
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 **Detect overflows** drop-down list, select `signed-and-unsigned`.
5. Polyspace detects a red **Overflow** error in the computation `num_items * sizeof(int)`.

See Also

Polyspace Analysis Options

`Detect overflows (-scalar-overflows-checks)`

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:

- “Review Red Checks” on page 8-10
- “Review Orange Checks” on page 8-17

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 “Constrain Stubbed Functions” on page 5-44.

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 “Add Review Comments to Results” on page 8-32 and “Justify Results Through Code Annotations” on page 8-36.

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.

A red or orange **Uncaught exception** check can arise due to the following reasons.

Message in Result Details	Description
Function throws or call to function throws.	The function body contains a `throw` statement or a function call that leads to a `throw` statement. *Possible Fix:* Navigate to the function containing the `throw` statement. Catch the exception as early as possible by using a `try-catch` block.
Exception raised is not specified in the throw list.	The function header contains a `throw` declaration. The data types in the declaration do not match the data type in `throw` statements in the function body. *Possible Fix:* Change the data type in the `throw` declaration or the `throw` statements in the function body.

Step 2: Determine Root Cause of Check

If you do not catch an exception, it propagates up the function call hierarchy from the function where the exception originates to the `main` function. If you fix a red or orange **Uncaught exception** check in the function where the exception originates, the later **Uncaught exception** checks are also fixed.

Navigate to the **Uncaught exception** check in the function where the exception originates. You can start from an arbitrary **Uncaught exception** check on the **Source** pane in the Polyspace user interface.

- If the **Uncaught exception** check appears on a function definition, see the function header.

1. If the check appears on the function name in the header, find another function call in the body that contains a red or orange **Uncaught exception** check. If the check appears on the function return type in the header, you have already found the function where the exception originates.
2. If you find another function call with an **Uncaught exception** check, right-click the call and select **Go To Definition**. You go to one level down in the function call hierarchy to the function definition.

   If the option **Go To Definition** is not available, on the **Result Details** pane, select the *fx* icon. Use the **Call Hierarchy** pane to navigate the function call hierarchy.
3. Continue navigating down the call hierarchy until you find the function that contains a `throw` statement.

- If the **Uncaught exception** check appears on a function call:

  1. Right-click the call and select **Go To Definition**. You go to one level down in the function call hierarchy to the function definition.

     If the option **Go To Definition** is not available, on the **Result Details** pane, select the *fx* icon. Use the **Call Hierarchy** pane to navigate the function call hierarchy.
  2. Continue navigating down the call hierarchy until you find the function that contains a `throw` statement.

- If the **Uncaught exception** check appears on a `new` statement, navigate to the definition of the constructor that you are using for object creation. Use the same root cause navigation steps as earlier until you find the `throw` statement that causes the check.

  To navigate to the constructor definition from the `new` statement:

  1. Select the **Uncaught exception** check on the `new` statement.
  2. On the **Result Details** pane, select the *fx* icon.
  3. On the **Call Hierarchy** pane, double-click the constructor *`className`*`::`*`className`*.

Possible Fix: Catch the exception as early as possible.

- If the `throw` statement appears in the function body, place the statement in a `try-catch` block.
- You can also catch the exception one level up in the call hierarchy. Place the call to the function in a `try-catch` block.

  To navigate one level up in the call hierarchy, select the function name in the header.

  On the **Result Details** pane, select the *fx* icon. On the **Call Hierarchy** pane, select each caller denoted by ◂ .

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. To:

- Add justification in your result, see “Add Review Comments to Results” on page 8-32.
- Add justification in your code, see “Justify Results Through Code Annotations” on page 8-36.

In this section...

“Step 1: Interpret Check Information” on page 9-93
“Step 2: Determine Root Cause of Check” on page 9-94
“Step 3: Look for Common Causes of Check” on page 9-96

Step 1: Interpret Check Information

1. Select the check on the **Results List** or **Source** pane.
2. View the message on the **Result Details** pane.

   The message explains why the block of code is unreachable.

   ID 6: Unreachable code
   Unreachable code
   If-condition always evaluates to true at line 47 (column 8).
   Block ends at line 51 (column 4)

3. A code block is usually unreachable when the condition that determines entry into the block is not satisfied. See why the condition is not satisfied.

   a. On the **Source** pane, place your cursor on the variables involved in the condition to determine their values.
   b. Using these values, see why the condition is not satisfied.

   Note Sometimes, a condition itself is redundant. For example, it is always true or coupled:

- Through an `||` operator to another condition that is always true.
- Through an `&&` operator to another condition that is always false.

For example, in the following code, the condition `x%2==0` is redundant because the first condition `x>0` is always true.

```
assert(x>0);
if(x>0 || x%2 == 0)
```

If a condition is redundant, instead of a block of code, the condition itself is marked gray.

Step 2: Determine Root Cause of Check

Trace the data flow for each variable involved in the condition.

In the following example, trace the data flow for `arg`.

```
void foo(void) {
    int x=0;
    .
    .
    bar(x);
    .
    .
}

void bar(int arg) {
    if(arg==0) {
        /*Block 1*/
    }
    else {
        /*Block 2*/
    }
}
```

You might find that `bar` is called only from `foo`. Since the only argument of `bar` has value 0, the `else` branch of `if(arg==0)` is unreachable.

Possible fix: If you do not intend to call `bar` elsewhere and know that the values passed to `bar` will not change, you can remove the `if-else` statement in `bar` and retain only the content of `Block 1`.

To trace the data flow, select the check and note the information on the **Result Details** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Result Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.
- Otherwise, for each variable involved in the condition, find previous instances and trace back to the root cause of check. For more information on common root causes, see "Step 3: Look for Common Causes of Check" on page 9-96.

  Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 Double-click the variable on the **Source** pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.

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.	**1** Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. **2** On this pane, select the previous instances of the variable. Write operations on the variable are indicated with and read operations with .
Function return value `ret=func();`	**1** Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration. **2** In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 9-104.

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 8-74

  An exception to this behavior is overflow. If you specify the appropriate **Overflow computation mode**, Polyspace wraps the result of an overflow and does not terminate the execution paths. See `Overflow computation mode (-scalar-overflows-behavior)`.
- "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:

- “Review Red Checks” on page 8-10
- “Review Orange Checks” on page 8-17

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 9-96.

  Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 Double-click the variable on the **Source** pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with and read operations with .
Function return value `ret=func();`	1 Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration. 2 In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 9-104.

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 9-63.
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 “Add Review Comments to Results” on page 8-32 and “Justify Results Through Code Annotations” on page 8-36.

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 10-19.
3. Try to reduce your code complexity and rerun the verification. Otherwise, add a comment and a justification in your result or code describing why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Note Before justifying an orange check, consider carefully whether you can improve your coding design.

Find Relations Between Variables in Code

This tutorial shows how to determine if the variables in an arbitrary operation in your code are previously related.

For instance, consider this operation:

```
return(var1 - var2);
```

- In your IDE, you can place breakpoints to stop execution and determine the values of `var1` and `var2` *for a specific run*.
- In Polyspace, after you analyze your code, the tooltips on `var1` and `var2` show their range of values *for all runs* that the verification considers.

However, the range information is not enough to determine if the variables are related. You must perform additional steps to determine the relation.

Insert Pragma to Determine Variable Relation

In this example, consider the operation highlighted. You cannot tell from a quick glance if `wheel_speed` and `wheel_speed_old` are related. However, this information is crucial to understand a possible bug in subsequent operations.

```
#define MAX_SPEED 120
#define TEST_TIME 10000

int wheel_speed;
int wheel_speed_old;

int out;

int update_speed(int new_speed) {
    if(new_speed < MAX_SPEED)
    return new_speed;
    else
    return MAX_SPEED;
}

void increase_speed(void)
{

  int temp, index=1;
```

```
  while(index<TEST_TIME) {
    temp = wheel_speed - wheel_speed_old;

    if(index > 1) {
       if (temp < 0)
         out = 1;
       else
         out = 0;
    }

    wheel_speed_old = update_speed(wheel_speed);
    index++;
  }

}
```

To understand why you need the relation between `wheel_speed` and `wheel_speed_old` and how to find the relation:

1. Constrain the range of the variable `wheel_speed` to 0..100 for the Polyspace analysis. Use the analysis option `Constraint setup (-data-range-specifications)`.
2. Run analysis on this code and open the results. Select the gray **Unreachable code** check.

   ```
   if (temp < 0)
            out = 1;
   ```

   The check indicates that the variable `temp` is nonnegative. `temp` comes from the previous operation:

   ```
   temp = wheel_speed - wheel_speed_old;
   ```
3. See the range of `wheel_speed` and `wheel_speed_old`. Place your cursor on these variables. You see these ranges:

   - `wheel_speed`: 0..100
   - `wheel_speed_old`: Full range of an `int` variable.

   It is not clear from these ranges why `wheel_speed - wheel_speed_old` is always nonnegative. You have to find out if the variables are somehow related.

4 Insert a pragma before the line where you want the variable relation. Add the following line just before `if(temp < 0)`:

```
#pragma Inspection_Point wheel_speed wheel_speed_old
```

5 Rerun the analysis and open the results. Place your cursor on `wheel_speed_old` in the line that you added.

   The tooltip confirms that `wheel_speed` and `wheel_speed_old` are related:

```
wheel_speed_old <= wheel_speed
```

6 To find how the two variables got related, search for all instances of `wheel_speed_old`. On the **Source** pane, right-click `wheel_speed_old` and select **Search For All References**.

   Browse through the instances. In this case, you see that the line which relates `wheel_speed` and `wheel_speed_old` is:

```
  wheel_speed_old = update_speed(wheel_speed);
```

   This line ensures that after the first run of the `while` loop, `wheel_speed_old` is less than or equal to `wheel_speed_old`. The branch `if(index > 1)` is entered from the second run onwards. In this branch, the relation between `wheel_speed` and `wheel_speed_old` is reflected through the gray **Unreachable code** check.

Tip Ignore the details of the relation shown in the tooltip. Use the tooltip to confirm if certain variables are related. Then, search for instances of the variable to find how they are related.

Further Exploration

You can use the pragma `Inspection_Point` to determine the relation between variables at any point in the code. You can enter as many variables as you want in the `#pragma` statement:

```
#pragma Inspection_Point var1 var2 ... varn
```

Try this technique on other examples. For instance, select **Help** > **Examples** > **Code_Prover_Example.psprj**. Group the results by file. In the file `single_file_analysis.c`, you see this code:

```
if (output_v7 >= 0) {

  #pragma Inspection_Point output_v7 s8_ret
  saved_values[output_v7] = s8_ret;
  return s8_ret;
}
```

If you place your cursor on `s8_ret` in the last two statements, you find that the ranges of `s8_ret` are different. Find out what changed between the two statements.

Hint: The tooltip in the `#pragma` statement indicates that the variable `s8_ret` is related to the variable `output_v7`. Note the orange check that reduces the range of `output_v7`.

See Also

Related Examples

- “Review Red Checks” on page 8-10
- “Review Gray Checks” on page 8-15
- “Review Orange Checks” on page 8-17

Managing Orange Checks

- “Sources of Orange Checks” on page 10-2
- “Managing Orange Checks” on page 10-5
- “Limit Display of Orange Checks” on page 10-10
- “Critical Orange Checks” on page 10-13
- “Reduce Orange Checks” on page 10-16
- “Test Orange Checks for Run-Time Errors” on page 10-19
- “Limitations of Automatic Orange Tester” on page 10-24

Sources of Orange Checks

In this section...
"When Orange Checks Occur" on page 10-2 "Why Review Orange Checks" on page 10-2 "How to Review Orange Checks" on page 10-3 "How to Reduce Orange Checks" on page 10-3

When Orange Checks Occur

An orange check indicates that Polyspace detects a possible run-time error but cannot prove it. A check on an operation appears orange if both conditions are true:

First condition	Second condition	Example
The operation occurs multiple times on an execution path or on multiple execution paths.	During static verification, the operation fails only a fraction of times or only on a fraction of paths.	The operation occurs in: • A loop with more than one iterations. • A function that is called more than once.
The operation involves a variable that can take multiple values.	During static verification, the operation fails only for a fraction of values.	The operation involves a `volatile` variable.

During static verification, Polyspace can consider more execution paths than the execution paths that occur during run time. If an operation fails on a subset of paths, Polyspace cannot determine if that subset actually occurs during run time. Therefore, instead of a red check, it produces an orange check on the operation.

Why Review Orange Checks

Considering a superset of actual execution paths is a sound approximation because Polyspace does not lose information. If an operation contains a run-time error, Polyspace does not produce a green check on the operation. If Polyspace cannot prove the run-time error because of approximations, it produces an orange check. Therefore, you must review orange checks.

Examples of Polyspace approximations include:

- Approximating the range of a variable at a certain point in the execution path. For instance, Polyspace can approximate the range `{-1} U [0,10]` of a `float` variable by `[-1,10]`.
- Approximating the interleaving of instructions in multitasking code. For instance, even if certain instructions in a pair of tasks cannot interrupt each other, Polyspace verification might not take that into account.

How to Review Orange Checks

To ensure that an operation does not fail during run time:

1. Determine if the execution paths on which the operation fails can actually occur.

   For more information, see "Review Orange Checks" on page 8-17.
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. For more information on:

   - Entering comments in results, see "Add Review Comments to Results" on page 8-32.
   - Entering comments in code, see"Justify Results Through Code Annotations" on page 8-36.

   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 10-16.
- 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 10-17.

- 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 10-17.

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**.

In this section...
“Software Development Stage” on page 10-6
“Quality Goals” on page 10-9

Software Development Stage

Development Stage	Situation	Review Process
Initial stage or unit development stage	In initial stages of development, you can have partially developed code or want to verify each source file independently. In that case, it is possible that: • You have not defined all your functions and class methods. • You do not have a `main` function Because of insufficient information in the code, Polyspace makes assumptions that result in many orange checks. For instance, if you use the default configuration, Polyspace assumes full range for inputs of functions that are not called in the code.	In the initial stages of development, review all red checks. For orange checks, depending on your requirements, do one of the following: • You want your partially developed code to be free of errors independent of the remaining code. For instance, you want your functions to not cause run-time errors for any input. If so, review orange checks at this stage. • You might want your partially developed code to be free of errors only in the context of the remaining code. If so, do one of the following: • Ignore orange checks at this stage. • Provide the context and then review orange checks. For instance, you can provide stubs for undefined functions to emulate them more accurately.

Development Stage	Situation	Review Process
		For more information, see “Provide Context for Verification” on page 10-16.
Later stage or integration stage	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.	Depending on the time you want to spend, do one of the following: • Review red checks only. • Review red and critical orange checks.

Development Stage	Situation	Review Process
Final stage	• You have provided all your source files. • You have emulated run-time environment accurately through the verification options.	Depending on the time you want to spend, do one of the following: • Review red checks and critical orange checks. • Review red checks and all orange checks. For each orange check: • If the check indicates a run-time error, fix the cause of the error. • If the check indicates a Polyspace approximation, enter a comment in your results or source code. As part of your final release process, you can have one of these criteria: • All red and critical orange checks must be reviewed and justified. • All red and orange checks must be reviewed and justified. To justify a check, assign the **Status** of `No action planned` or `Justified` to the check.

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

- “Prioritize Check Review” on page 8-119

More About

- “Sources of Orange Checks” on page 10-2

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 10-13.
- Limit the number or suppress orange checks for certain check types, using additional options on drop-down list.

  You can add predefined options to the list or create your own options. If you create your own options, you can share the option files to help developers in your organization review at least a certain number or percentage of orange checks.

1 Select **Tools** > **Preferences**.
2 On the **Review Scope** tab, do one of the following:

   - To add predefined options to the drop-down list on the **Results List** pane, select **Include Quality Objectives Scopes**.

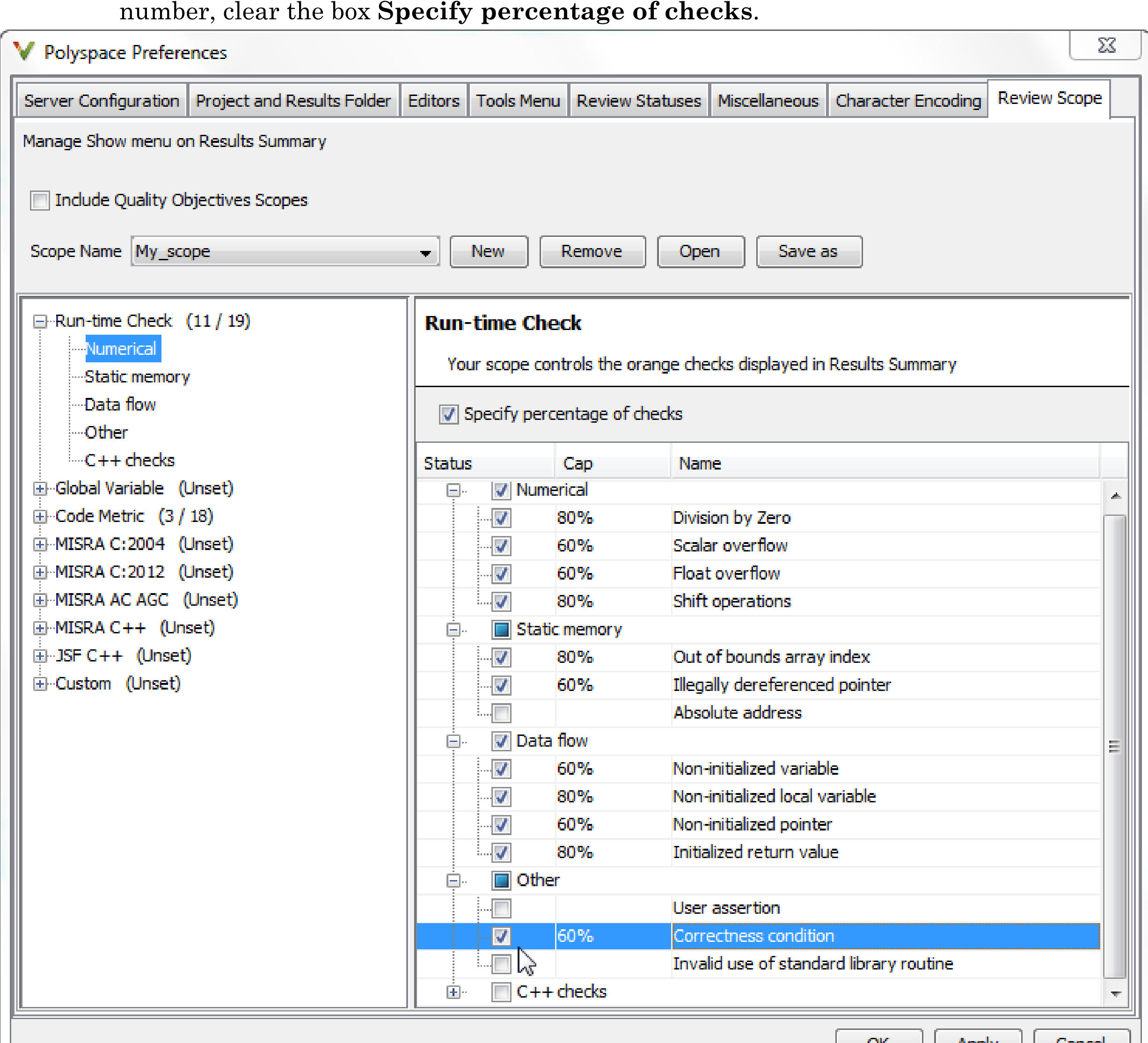
     The **Scope Name** list shows additional options, `HIS`, `SQO-4`, `SQO-5` and `SQO-6`. Select an option to see the limit values.

     In addition to orange checks, the options impose limits on the display of code metrics and coding rule violations. The option `HIS` displays code metrics only. For a detailed explanation of the predefined options, see “Software Quality Objectives” on page 8-122.
   - 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 “Add Review Comments to Results” on page 8-32.

See Also

Related Examples

- “Prioritize Check Review” on page 8-119
- “Filter and Group Results” on page 8-113
- “Reduce Orange Checks” on page 10-16

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.

In this section...
"Path" on page 10-13
"Bounded Input Values" on page 10-14
"Unbounded Input Values" on page 10-15

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: [-2^31 .. 2^31-1]
```

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 10-10.

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.

To provide more context for verification and reduce orange checks, use the following methods.

Method	Example
Define how the `main` generated by Polyspace initializes variables and calls functions	“Code Prover Verification”
Define ranges for global variables and function arguments.	“Create Constraint Template After Analysis” on page 5-36

Method	Example
Define execution sequence for multitasking code.	"Manually Model Scheduling of Tasks" on page 5-76
Map an imprecisely analyzed function to a standard function for precise results at the function call sites.	`-function-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 (-O)`	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**	• "Software Quality Objective Subsets (C: 2004)" on page 11-5 • "Software Quality Objective Subsets (C: 2012)" on page 11-58
Generated C code	**Check MISRA AC AGC**	"Software Quality Objective Subsets (AC AGC)" on page 11-10
Handwritten C++ code	**Check MISRA C++ rules**	"Software Quality Objective Subsets (C++)" on page 11-85

b. From the option drop-down list, select `SQO-subset1` or `SQO-subset2`.

2. Run verification and review your results.
3. Fix the coding rule violations.

See Also

More About

- "Sources of Orange Checks" on page 10-2
- "Managing Orange Checks" on page 10-5

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 "Sources of Orange Checks" on page 10-2.

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);
}
```

In this section...
"Run Tests for Full Range of Input" on page 10-19
"Run Tests for Specified Range of Input" on page 10-22

Run Tests for Full Range of Input

Note The Automatic Orange Tester is not supported on Mac.

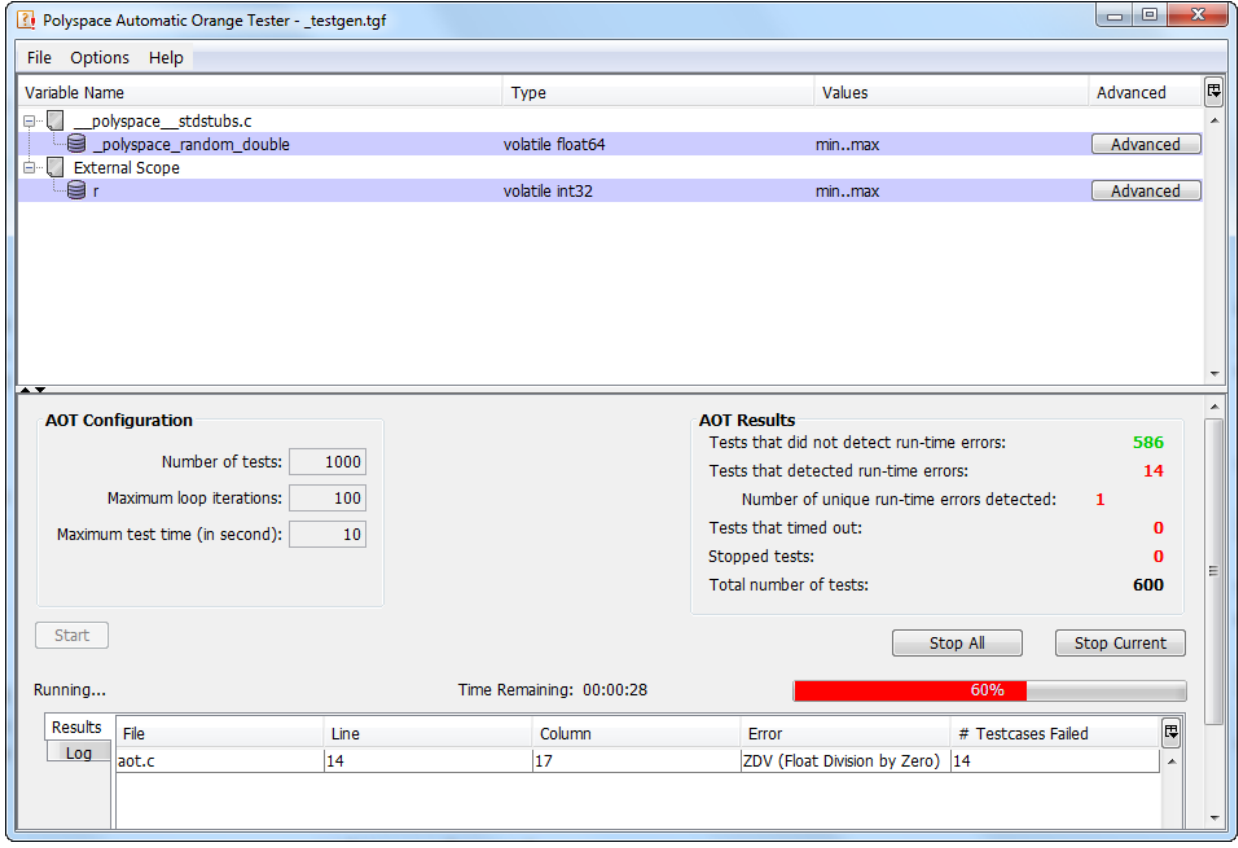
---

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

- "Prioritize Check Review" on page 8-119
- "Identify Function Call with Run-Time Error" on page 9-63

More About

- "Limitations of Automatic Orange Tester" on page 10-24
- "Sources of Orange Checks" on page 10-2
- "Managing Orange Checks" on page 10-5

Limitations of Automatic Orange Tester

The Automatic Orange Tester has the following limitations:

In this section...
"Unsupported Platforms" on page 10-24
"Unsupported Polyspace Options" on page 10-24
"Options with Restrictions" on page 10-24
"Unsupported C Routines" on page 10-25

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-16its`
- `-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`
- `sigrelse`
- `sigpause`
- `sigignore`
- `sigaction`
- `sigpending`
- `sigsuspend`
- `sigvec`
- `sigblock`
- `sigsetmask`
- `sigprocmask`
- `siginterrupt`
- `srand`
- `srandom`
- `initstate`
- `setstate`

11

Coding Rule Sets and Concepts

- "Rule Checking" on page 11-2
- "Polyspace MISRA C 2004 and MISRA AC AGC Checkers" on page 11-4
- "Software Quality Objective Subsets (C:2004)" on page 11-5
- "Software Quality Objective Subsets (AC AGC)" on page 11-10
- "MISRA C:2004 and MISRA AC AGC Coding Rules" on page 11-14
- "Polyspace MISRA C:2012 Checker" on page 11-56
- "Software Quality Objective Subsets (C:2012)" on page 11-58
- "Coding Rule Subsets Checked Early in Analysis" on page 11-63
- "Unsupported MISRA C:2012 Guidelines" on page 11-83
- "Polyspace MISRA C++ Checker" on page 11-84
- "Software Quality Objective Subsets (C++)" on page 11-85
- "MISRA C++ Coding Rules" on page 11-92
- "Polyspace JSF C++ Checker" on page 11-120
- "JSF C++ Coding Rules" on page 11-121

Rule Checking

Polyspace Coding Rule Checker

Polyspace software allows you to analyze code to demonstrate compliance with established C and C++ coding standards:

- MISRA C 2004 on page 11-4
- MISRA C 2012 on page 11-56
- MISRA C++:2008 on page 11-84
- JSF++:2005 on page 11-121

Applying coding rules can reduce the number of defects and improve the quality of your code.

While creating a project, you specify both the coding standard, and which rules to enforce. Polyspace software performs rule checking before and during the analysis. Violations appear in the **Results List** pane.

If any source files in the analysis do 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

---

Note When you enable the Compilation Assistant *and* coding rules checking, the software does not report coding rule violations if there are compilation errors.

Differences Between Bug Finder and Code Prover

Coding rule checker results can differ between Polyspace Bug Finder and Polyspace Code Prover. The rule checking engines are identical in Bug Finder and Code Prover, but the context in which the checkers execute is not the same. If a project is launched from Bug Finder and Code Prover with the same source files and same configuration options, the coding rule results can differ. For example, the main generator used in Code Prover activates global variables, which causes the rule checkers to identify such global

variables as initialized. The Bug Finder does not have a main generator, so handles the initialization of the global variables differently. Another difference is how violations are reported. The coding rules violations found in header files are not reported to the user in Bug Finder, but these violations are visible in Code Prover.

This difference can occur in MISRA C:2004, MISRA C:2012, MISRA C++, and JSF++. See the **Polyspace Specification** column or the **Description** for each rule.

Even though there are differences between rules checkers in Bug Finder and Code Prover, both reports are valid in their own context. For quick coding rules checking, use Polyspace Bug Finder.

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.[2]

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 11-5
- "Software Quality Objective Subsets (AC AGC)" on page 11-10

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

---

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

Software Quality Objective Subsets (C:2004)

In this section...
“Rules in SQO-Subset1” on page 11-5
“Rules in SQO-Subset2” on page 11-6

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 *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
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 *for* statement shall not contain any objects of floating type.
13.5	The three expressions of a *for* statement shall be concerned only with loop control.
14.4	The *goto* statement shall not be used.
14.7	A function shall have a single point of exit at the end of the function.

Rule number	Description
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.
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	*typedefs* that indicate size and signedness should be used in place of the basic types
8.7	Objects shall be defined at block scope if they are only accessed from within a single function

Rule number	Description
8.11	The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
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
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

Rule number	Description
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 *for* statement shall not contain any objects of floating type.
13.5	The three expressions of a *for* statement shall be concerned only with loop control.
13.6	Numeric variables being used within a “*for”* loop for iteration counting should not be modified in the body of the loop
14.4	The *goto* statement shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
14.8	The statement forming the body of a *switch, while, do while* or *for* statement shall be a compound statement
14.10	All *if else if* constructs should contain a final *else* clause
15.3	The final clause of a *switch* statement shall be the *default* clause
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration
16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty

Rule number	Description
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.
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; }
```

Software Quality Objective Subsets (AC AGC)

In this section...

"Rules in SQO-Subset1" on page 11-10
"Rules in SQO-Subset2" on page 11-11

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 *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
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.
6.3	*typedefs* that indicate size and signedness should be used in place of the basic types
8.7	Objects shall be defined at block scope if they are only accessed from within a single function
8.11	The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by 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

Rule number	Description
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 ##
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.

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*.

MISRA C:2004 and MISRA AC AGC Coding Rules

In this section...
"Supported MISRA C:2004 and MISRA AC AGC Rules" on page 11-14
"Unsupported MISRA C:2004 and MISRA AC AGC Rules" on page 11-53

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.

- "Environment" on page 11-16
- "Language Extensions" on page 11-18
- "Documentation" on page 11-19
- "Character Sets" on page 11-19

- "Identifiers" on page 11-19
- "Types" on page 11-21
- "Constants" on page 11-22
- "Declarations and Definitions" on page 11-22
- "Initialization" on page 11-25
- "Arithmetic Type Conversion" on page 11-26
- "Pointer Type Conversion" on page 11-30
- "Expressions" on page 11-31
- "Control Statement Expressions" on page 11-35
- "Control Flow" on page 11-38
- "Switch Statements" on page 11-40
- "Functions" on page 11-41
- "Pointers and Arrays" on page 11-43
- "Structures and Unions" on page 11-44
- "Preprocessing Directives" on page 11-44
- "Standard Libraries" on page 11-49
- "Runtime Failures" on page 11-53

Environment

N.	MISRA Definition	Messages in report file	Polyspace Specification
1.1	All code shall conform to ISO 9899:1990 “Programming languages - C”, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.	The text *All code shall conform to ISO 9899:1990 Programming languages C, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996* precedes each of the following messages: • ANSI C does not allow ‘#include_next' • ANSI C does not allow macros with variable arguments list • ANSI C does not allow ‘#assert’ • ANSI C does not allow '#unassert' • ANSI C does not allow testing assertions • ANSI C does not allow '#ident' • ANSI C does not allow '#sccs' • text following '#else' violates ANSI standard. • text following '#endif' violates ANSI standard. • text following '#else' or '#endif' violates ANSI standard.	All the supported extensions lead to a violation of this MISRA rule. Standard compilation error messages do not lead to a violation of this MISRA rule and remain unchanged.

N.	MISRA Definition	Messages in report file	Polyspace Specification
1.1 (cont.)		The text *All code shall conform to ISO 9899:1990 Programming languages C, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/ COR2:1996* precedes each of the following messages: • ANSI C90 forbids 'long long int' type. • ANSI C90 forbids 'long double' type. • ANSI C90 forbids long long integer constants. • Keyword 'inline' should not be used. • Array of zero size should not be used. • Integer constant does not fit within unsigned long int. • Integer constant does not fit within long int. • Too many nesting levels of #includes: N_1. The limit is N_0. • Too many macro definitions: N_1. The limit is N_0. • Too many nesting levels for control flow: N_1. The limit is N_0.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
		• Too many enumeration constants: N_1. The limit is N_0.	

Language Extensions

N.	MISRA Definition	Messages in report file	Polyspace Specification
2.1	Assembly language shall be encapsulated and isolated.	Assembly language shall be encapsulated and isolated.	No warnings if code is encapsulated in the following: • `asm` functions or `asm pragma` • Macros
2.2	Source code shall only use /* */ style comments	C++ comments shall not be used.	C++ comments are handled as comments but lead to a violation of this MISRA rule **Note**: This rule cannot be annotated in the source code.
2.3	The character sequence /* shall not be used within a comment	The character sequence /* shall not appear within a comment.	This rule violation is also raised when the character sequence /* inside a C++ comment. **Note**: This rule cannot be annotated in the source code.

Documentation

Rule	MISRA Definition	Messages in report file	Polyspace Specification
3.4	All uses of the *#pragma* directive shall be documented and explained.	All uses of the #pragma directive shall be documented and explained.	To check this rule, you must list the pragmas that are allowed in source files by using the option `Allowed pragmas (-allowed-pragmas)`. If Polyspace finds a pragma not in the allowed pragma list, a violation is raised.

Character Sets

N.	MISRA Definition	Messages in report file	Polyspace Specification
4.1	Only those escape sequences which are defined in the ISO C standard shall be used.	\<character> is not an ISO C escape sequence Only those escape sequences which are defined in the ISO C standard shall be used.	
4.2	Trigraphs shall not be used.	Trigraphs shall not be used.	Trigraphs are handled and converted to the equivalent character but lead to a violation of the MISRA rule

Identifiers

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	• Local declaration of XX is hiding another identifier. • Declaration of parameter XX is hiding another identifier.	Assumes that rule 8.1 is not violated.
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

N.	MISRA Definition	Messages in report file	Polyspace Specification
5.7	No identifier name should be reused.	{identifier}'%s' should not be reused. (already used as {identifier} at %s:%d)	No violation reported when: • Different functions have parameters with the same name • Different functions have local variables with the same name • A function has a local variable that has the same name as a parameter of another function

Types

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	• 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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
6.3	*typedefs* that indicate size and signedness should be used in place of the basic types	typedefs that indicate size and signedness should be used in place of the basic types.	No warning is given in typedef definition.
6.4	Bit fields shall only be defined to be of type *unsigned int* or *signed int*.	Bit fields shall only be defined to be of type unsigned int or signed int.	
6.5	Bit fields of type *signed int* shall be at least 2 bits long.	Bit fields of type signed int shall be at least 2 bits long.	No warning on anonymous signed int bitfields of width 0 - Extended to all signed bitfields of size <= 1 (if Rule **6.4** is violated).

Constants

N.	MISRA Definition	Messages in report file	Polyspace Specification
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.	• Octal constants other than zero and octal escape sequences shall not be used. • Octal constants (other than zero) should not be used. • Octal escape sequences should not be used.	

Declarations and Definitions

N.	MISRA Definition	Messages in report file	Polyspace Specification
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	• Function XX has no complete prototype visible at call. • Function XX has no prototype visible at definition.	Prototype visible at call must be complete.

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	• 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	• Object 'XX' should not be defined in a header file. • Function 'XX' should not be defined in a header file. • Fragment of function should not be defined in a header file.	Tentative definitions are considered as definitions. For objects with file scope, tentative definitions are declarations that: • Do not have initializers. • Do not have storage class specifiers, or have the `static` specifier
8.6	Functions shall always be declared at file scope.	Function 'XX' should be declared at file scope.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
8.7	Objects shall be defined at block scope if they are only accessed from within a single function	Object 'XX' should be declared at block scope.	Restricted to static objects.
8.8	An external object or function shall be declared in one file and only one file	Function/Object 'XX' has external declarations in multiples files.	Restricted to explicit extern declarations (tentative definitions are ignored). Polyspace considers that variables or functions declared `extern` in a non-header file violate this rule. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8.9	Definition: An identifier with external linkage shall have exactly one external definition.	• Procedure/Global variable XX multiply defined. • Forbidden multiple tentative definitions for object XX • Global variable has multiple tentative definitions • Undefined global variable XX	The checker flags multiple definitions only if the definitions occur in different files. No warnings appear on predefined symbols. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8.10	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required	Function/Variable XX should have internal linkage.	Assumes that 8.1 is not violated. No warning if 0 uses. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

N.	MISRA Definition	Messages in report file	Polyspace Specification
8.11	The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage	static storage class specifier should be used on internal linkage symbol XX.	
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization	Size of array 'XX' should be explicitly stated.	

Initialization

N.	MISRA Definition	Messages in report file	Polyspace Specification
9.1	All automatic variables shall have been assigned a value before being used.		Checked during code analysis. Violations displayed as Non-initialized variable results. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
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 Specification
10.1	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: • it is not a conversion to a wider integer type of the same signedness, or • the expression is complex, or • the expression is not constant and is a function argument, or • the expression is not constant and is a return expression	• Implicit conversion of the expression of underlying type XX to the type XX that is not a wider integer type of the same signedness. • Implicit conversion of one of the binary operands whose underlying types are XX and XX • Implicit conversion of the binary right hand operand of underlying type XX to XX that is not an integer type. • Implicit conversion of the binary left hand operand of underlying type XX to XX that is not an integer type. • Implicit conversion of the binary right hand operand of underlying type XX to XX that is not a wider integer type of the same signedness or Implicit conversion of the binary ? left hand operand of underlying type XX to XX, but it is a complex expression. • Implicit conversion of complex integer expression of underlying type XX to XX.	ANSI C base types order (signed char, short, int, long) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1. The same interpretation is applied on the unsigned version of base types. An expression of bool or enum types has int as underlying type. Plain char may have signed or unsigned underlying type (depending on Polyspace target configuration or option setting). The underlying type of a simple expression of struct.bitfield is the base type used in the bitfield definition, the bitfield width is not token into account and it assumes that only signed | unsigned int are used for bitfield (Rule 6.4). This rule violation is not produced on operations involving pointers. No violation reported when: • The implicit conversion is a type widening, without change of signedness of integer

N.	MISRA Definition	Messages in report file	Polyspace Specification
		• 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.	• The expression is an argument expression or a return expression No violation reported when the following are true: • Implicit conversion applies to a constant expression and is a type widening, with a possible change of signedness of integer. • The conversion does not change the representation of the constant value or the result of the operation. • The expression is an argument expression or a return expression or an operand expression of a non-bitwise operator. Conversions of constants are not reported for these cases to avoid flagging too many violations. If the constant can be represented in both the original and converted type, the conversion is less of an issue.

N.	MISRA Definition	Messages in report file	Polyspace Specification
10.2	The value of an expression of floating type shall not be implicitly converted to a different type if • it is not a conversion to a wider floating type, or • the expression is complex, or • the expression is a function argument, or • the expression is a return expression	• 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.	ANSI C base types order (float, double) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1. No violation reported when: • The implicit conversion is a type widening • The expression is an argument expression or a return expression.

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	• 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 token into account and it assumes that only signed, unsigned int are used for bitfield (Rule 6.4).
10.4	The value of a complex expression of float type may only be cast to narrower floating type	Complex expression of XX type may only be cast to narrower floating type, however the destination type is XX.	ANSI C base types order (float, double) defines that T1 is narrower than T2 if T2 is on the right hand of T1 or T2 = T1.

N.	MISRA Definition	Messages in report file	Polyspace Specification
10.5	If the bitwise operator ~ and << are applied to an operand of underlying type *unsigned char* or *unsigned short*, the result shall be immediately cast to the underlying type of the operand	Bitwise [<<|~] is applied to the operand of underlying type [unsigned char| unsigned short], the result shall be immediately cast to the underlying type.	
10.6	The "U" suffix shall be applied to all constants of *unsigned* types	No explicit 'U suffix on constants of an unsigned type.	Warning when the type determined from the value and the base (octal, decimal or hexadecimal) is unsigned and there is no suffix `u` or `U`. For example, when the size of the `int` and `long int` data types is 32 bits, the coding rule checker will report a violation of rule 10.6 for the following line: `int a = 2147483648;` There is a difference between decimal and hexadecimal constants when `int` and `long int` are not the same size.

Pointer Type Conversion

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
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
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
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.	A cast should not be performed between a pointer to object type and a different pointer to object type.	
11.5	A cast shall not be performed that removes any *const* or *volatile* qualification from the type addressed by a pointer	A cast shall not be performed that removes any *const* or *volatile* qualification from the type addressed by a pointer	Extended to all conversions

Expressions

N.	MISRA Definition	Messages in report file	Polyspace Specification
12.1	Limited dependence should be placed on C's operator precedence rules in expressions	Limited dependence should be placed on C's operator precedence rules in expressions	
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits.	• The value of *'sym'* depends on the order of evaluation. • The value of volatile *'sym'* depends on the order of evaluation because of multiple accesses.	Rule 12.2 check assumes that no assignment in expressions that yield a Boolean values (rule 13.1). The expression is a simple expression of symbols. `i = i ++;` is a violation, but `tab[2] = tab[2]++;` is not a violation.

N.	MISRA Definition	Messages in report file	Polyspace Specification
12.3	The `sizeof` operator should not be used on expressions that contain side effects.	The `sizeof` operator should not be used on expressions that contain side effects.	No warning on volatile accesses
12.4	The right hand operand of a logical && or `||` operator shall not contain side effects.	The right hand operand of a logical && or `||` operator shall not contain side effects.	No warning on volatile accesses
12.5	The operands of a logical && or `||` shall be primary-expressions.	• operand of logical && is not a primary expression • operand of logical `||` is not a primary expression • The operands of a logical && or `||` shall be primary-expressions.	During preprocessing, violations of this rule are detected on the expressions in #if directives. Allowed exception on associatively (a && b && c), (a `||` b `||` c).

N.	MISRA Definition	Messages in report file	Polyspace Specification

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 !).	• 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 '?:'.
			The operand of a logical operator should be a Boolean data type. Although the C standard does not explicitly define the Boolean data type, the standard implicitly assumes the use of the Boolean data type.
			
			Some operators may return Boolean-like expressions, for example, `(var == 0)`.
			
			Consider the following code:
			
			`unsigned char flag;`
			`if (!flag)`
			
			The rule checker reports a violation of rule 12.6:
			
			`Operand of '!' logical operator should be effectively Boolean.`
			
			The operand `flag` is not a Boolean but an `unsigned char`.
			
			To be compliant with rule 12.6, the code must be rewritten either as
			
			`if (!( flag != 0))`
			
			or
			
			`if (flag == 0)`
			
			The use of the option `-boolean-types` may

N.	MISRA Definition	Messages in report file	Polyspace Specification
			increase or decrease the number of warnings generated.
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed	• [~/Left Shift/Right shift/&] operator applied on an expression whose underlying type is signed. • Bitwise ~ on operand of signed underlying type XX. • Bitwise [<<|>>] on left hand operand of signed underlying type XX. • Bitwise [& | ^] on two operands of s	The underlying type for an integer is signed when: • it does not have a u or U suffix • it is small enough to fit into a 64 bits signed number
12.8	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.	• shift amount is negative • shift amount is bigger than 64 • Bitwise [<< >>] count out of range [0 ..X] (width of the underlying type XX of the left hand operand - 1)..	The numbers that are manipulated in preprocessing directives are 64 bits wide so that valid shift range is between 0 and 63 Check is also extended onto bitfields with the field width or the width of the base type when it is within a complex expression
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	• Unary - on operand of unsigned underlying type XX. • Minus operator applied to an expression whose underlying type is unsigned	The underlying type for an integer is signed when: • it does not have a u or U suffix • it is small enough to fit into a 64 bits signed number
12.10	The comma operator shall not be used.	The comma operator shall not be used.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
12.11	Evaluation of constant unsigned expression should not lead to wraparound.	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	

12.12	The underlying bit representations of floating-point values shall not be used.	The underlying bit representations of floating-point values shall not be used.	Warning when:
			• A float pointer is cast as a pointer to another data type. Casting a float pointer as a pointer to `void` does not generate a warning.
			• A float is packed with another data type. For example:
			`union {`
			`float f;`
			`int i;`
			`} …`

12.13	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression	Warning when ++ or -- operators are not used alone.

Control Statement Expressions

N.	MISRA Definition	Messages in report file	Polyspace Specification
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 Specification
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	No warning is given on integer constants. Example: if (2) The use of the option -`boolean-types` may increase or decrease the number of warnings generated.
13.3	Floating-point expressions shall not be tested for equality or inequality.	Floating-point expressions shall not be tested for equality or inequality.	Warning on directs tests only.
13.4	The controlling expression of a *for* statement shall not contain any objects of floating type	The controlling expression of a for statement shall not contain any objects of floating type	If *for* index is a variable symbol, checked that it is not a float.

N.	MISRA Definition	Messages in report file	Polyspace Specification
13.5	The three expressions of a *for* statement shall be concerned only with loop control	• 1st expression should be an assignment. • Bad type for loop counter (XX). • 2nd expression should be a comparison. • 2nd expression should be a comparison with loop counter (XX). • 3rd expression should be an assignment of loop counter (XX). • 3rd expression: assigned variable should be the loop counter (XX). • The following kinds of for loops are allowed: (a) all three expressions shall be present; (b) the 2nd and 3rd expressions shall be present with prior initialization of the loop counter; (c) all three expressions shall be empty for a deliberate infinite loop.	Checked if the for loop index (V) is a variable symbol; checked if V is the last assigned variable in the first expression (if present). Checked if, in first expression, if present, is assignment of V; checked if in 2nd expression, if present, must be a comparison of V; Checked if in 3rd expression, if present, must be an assignment of V.
13.6	Numeric variables being used within a *for* loop for iteration counting should not be modified in the body of the loop.	Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop.	Detect only direct assignments if the for loop index is known and if it is a variable symbol.

N.	MISRA Definition	Messages in report file	Polyspace Specification
13.7	Boolean operations whose results are invariant shall not be permitted	• Boolean operations whose results are invariant shall not be permitted. Expression is always true. • Boolean operations whose results are invariant shall not be permitted. Expression is always false. • Boolean operations whose results are invariant shall not be permitted.	During compilation, check comparisons with at least one constant operand. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results. • Bug Finder flags some violations of this rule through the `Dead code` and `Useless if` checkers. • Code Prover does not use gray code to flag violations of this rule.

Control Flow

N.	MISRA Definition	Messages in report file	Polyspace Specification
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: • have at least one side effect however executed, or • cause control flow to change	

N.	MISRA Definition	Messages in report file	Polyspace Specification
14.3	All non-null statements shall either • have at least one side effect however executed, or • cause control flow to change	A null statement shall appear on a line by itself	We assume that a ';' is a null statement when it is the first character on a line (excluding comments). The rule is violated when: • there are some comments before it on the same line. • there is a comment immediately after it • there is something else than a comment after the ';' on the same line.
14.4	The *goto* statement shall not be used.	The goto statement shall not be used.	
14.5	The *continue* statement shall not be used.	The continue statement shall not be used.	
14.6	For any iteration statement there shall be at most one *break* statement used for loop termination	For any iteration statement there shall be at most one break statement used for loop termination	
14.7	A function shall have a single point of exit at the end of the function	A function shall have a single point of exit at the end of the function	
14.8	The statement forming the body of a *switch, while, do while* or *for* statement shall be a compound statement	• The body of a do while statement shall be a compound statement. • The body of a for statement shall be a compound statement. • The body of a switch statement shall be a compound statement	

N.	MISRA Definition	Messages in report file	Polyspace Specification
14.9	An *if (expression)* construct shall be followed by a compound statement. The *else* keyword shall be followed by either a compound statement, or another *if* statement	• An if (expression) construct shall be followed by a compound statement. • The else keyword shall be followed by either a compound statement, or another if statement	
14.10	All *if else if* constructs should contain a final *else* clause.	All if else if constructs should contain a final else clause.	

Switch Statements

N.	MISRA Definition	Messages in report file	Polyspace Specification

15.0	Unreachable code is detected between switch statement and first case.
	
	Note This is not a MISRA C2004 rule.
		switch statements syntax normative restrictions.	Warning on declarations or any statements before the first switch case.
			
			Warning on label or jump statements in the body of switch cases.
			
			On the following example, the rule is displayed in the log file at line 3:
			
			`1 ...`
			`2 switch(index) {`
			`3  var = var + 1;`
			`// RULE 15.0`
			`// violated`
			`4case 1: ...`
			
			The code between switch statement and first case is checked as dead code by Polyspace. It follows ANSI standard behavior.

N.	MISRA Definition	Messages in report file	Polyspace Specification
15.1	A switch label shall only be used when the most closely-enclosing compound statement is the body of a *switch* statement	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	
15.2	An unconditional *break* statement shall terminate every non-empty switch clause	An unconditional break statement shall terminate every non-empty switch clause	Warning for each non-compliant case clause.
15.3	The final clause of a *switch* statement shall be the *default* clause	The final clause of a switch statement shall be the default clause	
15.4	A *switch* expression should not represent a value that is effectively Boolean	A switch expression should not represent a value that is effectively Boolean	The use of the option `-boolean-types` may increase the number of warnings generated.
15.5	Every *switch* statement shall have at least one *case* clause	Every switch statement shall have at least one case clause	

Functions

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	Done by Polyspace software (Use the call graph in Polyspace Code Prover). Polyspace also partially checks this rule during the compilation phase.
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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
16.4	The identifiers used in the declaration and definition of a function shall be identical.	The identifiers used in the declaration and definition of a function shall be identical.	Assumes that rules **8.8**, **8.1** and **16.3** are not violated. All occurrences are detected.
16.5	Functions with no parameters shall be declared with parameter type *void*.	Functions with no parameters shall be declared with parameter type void.	Definitions are also checked.
16.6	The number of arguments passed to a function shall match the number of parameters.	• Too many arguments to XX. • Insufficient number of arguments to XX.	Assumes that rule **8.1** is not violated.
16.7	A pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.	Pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.	Warning if a non-`const` pointer parameter is either not used to modify the addressed object or is passed to a call of a function that is declared with a `const` pointer parameter.
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Missing return value for non-void function XX.	Warning when a non-void function is not terminated with an unconditional return with an expression.
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty.	Function identifier XX should be preceded by a & or followed by a parameter list.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	Warning if a non-`void` function is called and the returned value is ignored. No warning if the result of the call is cast to `void`. No check performed for calls of `memcpy`, `memmove`, `memset`, `strcpy`, `strncpy`, `strcat`, or `strncat`.

Pointers and Arrays

N.	MISRA Definition	Messages in report file	Polyspace Specification
17.1	Pointer arithmetic shall only be applied to pointers that address an array or array element.	Pointer arithmetic shall only be applied to pointers that address an array or array element.	
17.2	Pointer subtraction shall only be applied to pointers that address elements of the same array	Pointer subtraction shall only be applied to pointers that address elements of the same array.	
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	
17.4	Array indexing shall be the only allowed form of pointer arithmetic.	Array indexing shall be the only allowed form of pointer arithmetic.	Warning on: • Operations on pointers. (`p+I`, `I+p`, and `p-I`, where `p` is a pointer and `I` an integer). • Array indexing on nonarray pointers.

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.

Structures and Unions

N.	MISRA Definition	Messages in report file	Polyspace Specification
18.1	All structure or union types shall be complete at the end of a translation unit.	All structure or union types shall be complete at the end of a translation unit.	Warning for all incomplete declarations of structs or unions.
18.2	An object shall not be assigned to an overlapping object.	• An object shall not be assigned to an overlapping object. • Destination and source of XX overlap, the behavior is undefined.	
18.4	Unions shall not be used	Unions shall not be used.	

Preprocessing Directives

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.1	#include statements in a file shall only be preceded by other preprocessors directives or comments	#include statements in a file shall only be preceded by other preprocessors directives or comments	A message is displayed when a #include directive is preceded by other things than preprocessor directives, comments, spaces or “new lines”.

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.2	Nonstandard characters should not occur in header file names in #include directives	• A message is displayed on characters ', " or /* between < and > in #include <filename> • A message is displayed on characters ', or /* between " and " in #include "filename"	
19.3	The *#include* directive shall be followed by either a <filename> or "filename" sequence.	• '#include' expects "FILENAME" or <FILENAME> • '#include_next' expects "FILENAME" or <FILENAME>	

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.	Macro '<name>' does not expand to a compliant construct.	We assume that a macro definition does not violate this rule when it expands to: • a braced construct (not necessarily an initializer) • a parenthesized construct (not necessarily an expression) • a number • a character constant • a string constant (can be the result of the concatenation of string field arguments and literal strings) • the following keywords: typedef, extern, static, auto, register, const, volatile, __asm__ and __inline__ • a do-while-zero construct
19.5	Macros shall not be #defined and #undefd within a block.	• Macros shall not be `#define`’d within a block. • Macros shall not be `#undef`’d within a block.	
19.6	#undef shall not be used.	`#undef` shall not be used.	
19.7	A function should be used in preference to a function like-macro.	A function should be used in preference to a function like-macro	Message on all function-like macro definitions.

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.8	A function-like macro shall not be invoked without all of its arguments	• arguments given to macro '<name>' • macro '<name>' used without args. • macro '<name>' used with just one arg. • macro '<name>' used with too many (<number>) args.	
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	Macro argument shall not look like a preprocessing directive.	This rule is detected as violated when the '#' character appears in a macro argument (outside a string or character constant)
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.	Parameter instance shall be enclosed in parentheses.	If `x` is a macro parameter, the following instances of x as an operand of the # and ## operators do not generate a warning: `#x`, `##x`, and `x##`. Otherwise, parentheses are required around `x`. The software does not generate a warning if a parameter is reused as an argument of a function or function-like macro. For example, consider a parameter x. The software does not generate a warning if `x` appears as `(x)` or `(x,` or `,x)` or `,x,`.

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.	Precautions shall be taken in order to prevent multiple inclusions.	When a header file is formatted as, #ifndef <control macro> #define <control macro> <contents> #endif or, #ifndef <control macro> #error ... #else #define <control macro> <contents> #endif it is assumed that precautions have been taken to prevent multiple inclusions. Otherwise, a violation of this MISRA rule is detected.

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	directive is not syntactically meaningful.	
19.17	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	• '#elif' not within a conditional. • '#else' not within a conditional. • '#elif' not within a conditional. • '#endif' not within a conditional. • unbalanced '#endif'. • unterminated '#if' conditional. • unterminated '#ifdef' conditional. • unterminated '#ifndef' conditional.	

Standard Libraries

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.	• The macro '<name> shall not be redefined. • The macro '<name> shall not be undefined.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.2	The names of standard library macros, objects and functions shall not be reused.	Identifier XX should not be used.	In case a macro whose name corresponds to a standard library macro, object or function is defined, the rule that is detected as violated is **20.1**. Tentative definitions are considered as definitions. For objects with file scope, tentative definitions are declarations that: • Do not have initializers. • Do not have storage class specifiers, or have the `static` specifier

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.3	The validity of values passed to library functions shall be checked.	Validity of values passed to library functions shall be checked	Warning for argument in library function call if the following are all true: • Argument is a local variable • Local variable is not tested between last assignment and call to the library function • Library function is a common mathematical function • Corresponding parameter of the library function has a restricted input domain. The library function can be one of the following : `sqrt`, `tan`, `pow`, `log`, `log10`, `fmod`, `acos`, `asin`, `acosh`, `atanh`, or `atan2`.
20.4	Dynamic heap memory allocation shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the dynamic heap memory allocation functions are actually macros and the macro is expanded in the code, this rule is detected as violated. Assumes rule **20.2** is not violated.
20.5	The error indicator errno shall not be used	The error indicator errno shall not be used	Assumes that rule **20.2** is not violated
20.6	The macro *offsetof*, in library <stddef.h>, shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	Assumes that rule **20.2** is not violated

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.7	The *setjmp* macro and the *longjmp* function shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the longjmp function is actually a macro and the macro is expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.8	The signal handling facilities of <signal.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case some of the signal functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.9	The input/output library <stdio.h> shall not be used in production code.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the input/output library functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.10	The library functions atof, atoi and atoll from library <stdlib.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the atof, atoi and atoll functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.11	The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the abort, exit, getenv and system functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.12	The time handling functions of library <time.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the time handling functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated

Runtime Failures

N.	MISRA Definition	Messages in report file	Polyspace Specification
21.1	Minimization of runtime failures shall be ensured by the use of at least one of: • static verification tools/ techniques; • dynamic verification tools/ techniques; • explicit coding of checks to handle runtime faults.		Done by Polyspace. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

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 "**Polyspace Specification**" column describes the reason each rule is not checked.

Environment

Rule	Description	Polyspace Specification
1.2 (Required)	No reliance shall be placed on undefined or unspecified behavior	Not statically checkable unless the data dynamic properties is taken into account
1.3 (Required)	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the language/compilers/ assemblers conform.	It is a process rule method.

Rule	Description	Polyspace Specification
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	Polyspace Specification
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	Polyspace Specification
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.

Rule	Description	Polyspace Specification
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	Polyspace Specification
18.3 (Required)	An area of memory shall not be reused for unrelated purposes.	"purpose" is functional design issue.

Polyspace MISRA C:2012 Checker

The Polyspace MISRA C:2012 checker helps you to comply with the MISRA C 2012 coding standard.[3]

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 Directive 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 11-58.

See Also

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

Related Examples

- “Set Up Coding Rules Checking” on page 12-2

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

More About

- “MISRA C:2012 Directives and Rules”
- “Software Quality Objective Subsets (C:2012)” on page 11-58

Software Quality Objective Subsets (C:2012)

In this section...
"Guidelines in SQO-Subset1" on page 11-58
"Guidelines in SQO-Subset2" on page 11-59

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.

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

Rule	Description
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
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

Rule	Description
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
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

Rule	Description
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)` | `Use generated code requirements (-misra3-agc-mode)`

Related Examples

- "Set Up Coding Rules Checking" on page 12-2

More About

- "MISRA C:2012 Directives and Rules"

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 for `Check MISRA C:2004 (-misra2)`, `Check MISRA AC AGC (-misra-ac-agc)`, and `Check MISRA C:2012 (-misra3)`.

Argument	Purpose
`single-unit-rules`	Check rules that apply only to single translation units.
`system-decidable-rules`	Check rules in the `single-unit-rules` subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit.

To run analysis only up to the compilation phase, use the option `Verification level (-to)`.

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	`typedef`s that indicate size and signedness should be used in place of the basic types.
6.4	Bit fields shall only be defined to be of type `unsigned int` or `signed int`.
6.5	Bit fields of type `signed int` shall be at least 2 bits long.

Constants

Rule	Description
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and Definitions

Rule	Description
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated.
8.3	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.
8.4*	If objects or functions are declared more than once their types shall be compatible.
8.5	There shall be no definitions of objects or functions in a header file.
8.6	Functions shall always be declared at file scope.
8.7	Objects shall be defined at block scope if they are only accessed from within a single function.
8.8*	An external object or function shall be declared in one file and only one file.
8.9*	An identifier with external linkage shall have exactly one external definition.
8.10*	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.
8.11	The `static` storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

Initialization

Rule	Description
9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.
9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Arithmetic Type Conversion

Rule	Description
10.1	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: • It is not a conversion to a wider integer type of the same signedness, or • The expression is complex, or • The expression is not constant and is a function argument, or • The expression is not constant and is a return expression
10.2	The value of an expression of floating type shall not be implicitly converted to a different type if • It is not a conversion to a wider floating type, or • The expression is complex, or • The expression is a function argument, or • The expression is a return expression
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.
10.4	The value of a complex expression of float type may only be cast to narrower floating type.
10.5	If the bitwise operator ~ and << are applied to an operand of underlying type `unsigned char` or `unsigned short`, the result shall be immediately cast to the underlying type of the operand
10.6	The "U" suffix shall be applied to all constants of `unsigned` types.

Pointer Type Conversion

Rule	Description
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to `void`.
11.3	A cast should not be performed between a pointer type and an integral type.
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.
11.5	A cast shall not be performed that removes any `const` or `volatile` qualification from the type addressed by a pointer

Expressions

<table>
<tr><th>Rule</th><th>Description</th></tr>
<tr><td>12.1</td><td>Limited dependence should be placed on C's operator precedence rules in expressions.</td></tr>
<tr><td>12.3</td><td>The <code>sizeof</code> operator should not be used on expressions that contain side effects.</td></tr>
<tr><td>12.5</td><td>The operands of a logical <code>&&</code> or <code>||</code> shall be primary-expressions.</td></tr>
<tr><td>12.6</td><td>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>).</td></tr>
<tr><td>12.7</td><td>Bitwise operators shall not be applied to operands whose underlying type is signed.</td></tr>
<tr><td>12.9</td><td>The unary minus operator shall not be applied to an expression whose underlying type is unsigned.</td></tr>
<tr><td>12.10</td><td>The comma operator shall not be used.</td></tr>
<tr><td>12.11</td><td>Evaluation of constant unsigned expression should not lead to wraparound.</td></tr>
<tr><td>12.12</td><td>The underlying bit representations of floating-point values shall not be used.</td></tr>
<tr><td>12.13</td><td>The increment (<code>++</code>) and decrement (<code>--</code>) operators should not be mixed with other operators in an expression</td></tr>
</table>

Control Statement Expressions

Rule	Description
13.1	Assignment operators shall not be used in expressions that yield Boolean values.
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a `for` statement shall not contain any objects of floating type.
13.5	The three expressions of a `for` statement shall be concerned only with loop control.
13.6	Numeric variables being used within a `for` loop for iteration counting should not be modified in the body of the loop.

Control Flow

Rule	Description
14.3	All non-null statements shall either • have at least one side effect however executed, or • cause control flow to change.
14.4	The `goto` statement shall not be used.
14.5	The `continue` statement shall not be used.
14.6	For any iteration statement, there shall be at most one `break` statement used for loop termination.
14.7	A function shall have a single point of exit at the end of the function.
14.8	The statement forming the body of a `switch`, `while`, `do while` or `for` statement shall be a compound statement.
14.9	An `if` (expression) construct shall be followed by a compound statement. The `else` keyword shall be followed by either a compound statement, or another `if` statement.
14.10	All `if else if` constructs should contain a final `else` clause.

Switch Statements

Rule	Description
15.0	Unreachable code is detected between `switch` statement and first `case`.
15.1	A `switch` label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement
15.2	An unconditional `break` statement shall terminate every non-empty `switch` clause.
15.3	The final clause of a `switch` statement shall be the `default` clause.
15.4	A `switch` expression should not represent a value that is effectively Boolean.
15.5	Every `switch` statement shall have at least one `case` clause.

Functions

Rule	Description
16.1	Functions shall not be defined with variable numbers of arguments.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.
16.4*	The identifiers used in the declaration and definition of a function shall be identical.
16.5	Functions with no parameters shall be declared with parameter type `void`.
16.6	The number of arguments passed to a function shall match the number of parameters.
16.8	All exit paths from a function with non-`void` return type shall have an explicit return statement with an expression.
16.9	A function identifier shall only be used with either a preceding `&`, or with a parenthesized parameter list, which may be empty.

Pointers and Arrays

Rule	Description
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	A type should not contain more than 2 levels of pointer indirection.

Structures and Unions

Rule	Description
18.1	All structure or union types shall be complete at the end of a translation unit.
18.4	Unions shall not be used.

Preprocessing Directives

Rule	Description
19.1	`#include` statements in a file shall only be preceded by other preprocessors directives or comments.
19.2	Nonstandard characters should not occur in header file names in `#include` directives.
19.3	The `#include` directive shall be followed by either a <filename> or "filename" sequence.
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.
19.5	Macros shall not be `#define`d and `#undef`d within a block.
19.6	`#undef` shall not be used.
19.7	A function should be used in preference to a function like-macro.
19.8	A function-like macro shall not be invoked without all of its arguments.
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
19.10	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of `#` or `##`.
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in `#ifdef` and `#ifndef` preprocessor directives and the `defined()` operator.
19.12	There shall be at most one occurrence of the `#` or `##` preprocessor operators in a single macro definition.
19.13	The `#` and `##` preprocessor operators should not be used.
19.14	The defined preprocessor operator shall only be used in one of the two standard forms.
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.

Rule	Description
19.17	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

Standard Libraries

Rule	Description
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.
20.2	The names of standard library macros, objects and functions shall not be reused.
20.4	Dynamic heap memory allocation shall not be used.
20.5	The error indicator errno shall not be used.
20.6	The macro offsetof, in library <stddef.h>, shall not be used.
20.7	The setjmp macro and the longjmp function shall not be used.
20.8	The signal handling facilities of <signal.h> shall not be used.
20.9	The input/output library <stdio.h> shall not be used in production code.
20.10	The library functions atof, atoi and atoll from library <stdlib.h> shall not be used.
20.11	The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.
20.12	The time handling functions of library <time.h> shall not be used.

The rules that are checked at a system level and appear only in the system-decidable-rules subset are indicated by an asterisk.

MISRA C: 2012 Rules

The software checks the following rules early in the analysis. The rules that are checked at a system level and appear only in the system-decidable-rules subset are indicated by an asterisk.

Standard C Environment

Rule	Description
1.1	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.
1.2	Language extensions should not be used.

Unused Code

Rule	Description
2.3*	A project should not contain unused type declarations.
2.4*	A project should not contain unused tag declarations.
2.5*	A project should not contain unused macro declarations.
2.6	A function should not contain unused label declarations.
2.7	There should be no unused parameters in functions.

Comments

Rule	Description
3.1	The character sequences `/*` and `//` shall not be used within a comment.
3.2	Line-splicing shall not be used in `//` comments.

Character Sets and Lexical Conventions

Rule	Description
4.1	Octal and hexadecimal escape sequences shall be terminated.
4.2	Trigraphs should not be used.

Identifiers

Rule	Description
5.1*	External identifiers shall be distinct.
5.2	Identifiers declared in the same scope and name space shall be distinct.
5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
5.4	Macro identifiers shall be distinct.
5.5	Identifiers shall be distinct from macro names.
5.6*	A typedef name shall be a unique identifier.
5.7*	A tag name shall be a unique identifier.
5.8*	Identifiers that define objects or functions with external linkage shall be unique.
5.9*	Identifiers that define objects or functions with internal linkage should be unique.

Types

Rule	Description
6.1	Bit-fields shall only be declared with an appropriate type.
6.2	Single-bit named bit fields shall not be of a signed type.

Literals and Constants

Rule	Description
7.1	Octal constants shall not be used.
7.2	A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type.
7.3	The lowercase character "l" shall not be used in a literal suffix.
7.4	A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

Declarations and Definitions

Rule	Description
8.1	Types shall be explicitly specified.
8.2	Function types shall be in prototype form with named parameters.
8.3*	All declarations of an object or function shall use the same names and type qualifiers.
8.4	A compatible declaration shall be visible when an object or function with external linkage is defined.
8.5*	An external object or function shall be declared once in one and only one file.
8.6*	An identifier with external linkage shall have exactly one external definition.
8.7*	Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.
8.8	The `static` storage class specifier shall be used in all declarations of objects and functions that have internal linkage.
8.9*	An object should be defined at block scope if its identifier only appears in a single function.
8.10	An inline function shall be declared with the `static` storage class.
8.11	When an array with external linkage is declared, its size should be explicitly specified.
8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.
8.14	The `restrict` type qualifier shall not be used.

Initialization

Rule	Description
9.2	The initializer for an aggregate or union shall be enclosed in braces.
9.3	Arrays shall not be partially initialized.
9.4	An element of an object shall not be initialized more than once.
9.5	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

The Essential Type Model

Rule	Description
10.1	Operands shall not be of an inappropriate essential type.
10.2	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.
10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.
10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.
10.5	The value of an expression should not be cast to an inappropriate essential type.
10.6	The value of a composite expression shall not be assigned to an object with wider essential type.
10.7	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.
10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

Pointer Type Conversion

Rule	Description
11.1	Conversions shall not be performed between a pointer to a function and any other type.
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type.
11.3	A cast shall not be performed between a pointer to object type and a pointer to a different object type.
11.4	A conversion should not be performed between a pointer to object and an integer type.
11.5	A conversion should not be performed from pointer to void into pointer to object.
11.6	A cast shall not be performed between pointer to void and an arithmetic type.
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type.
11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.
11.9	The macro `NULL` shall be the only permitted form of integer null pointer constant.

Expressions

Rule	Description
12.1	The precedence of operators within expressions should be made explicit.
12.3	The comma operator should not be used.
12.4	Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Side Effects

Rule	Description
13.3	A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.
13.4	The result of an assignment operator should not be used.
13.6	The operand of the `sizeof` operator shall not contain any expression which has potential side effects.

Control Statement Expressions

Rule	Description
14.4	The controlling expression of an `if` statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

Control Flow

Rule	Description
15.1	The `goto` statement should not be used.
15.2	The `goto` statement shall jump to a label declared later in the same function.
15.3	Any label referenced by a `goto` statement shall be declared in the same block, or in any block enclosing the `goto` statement.
15.4	There should be no more than one `break` or `goto` statement used to terminate any iteration statement.
15.5	A function should have a single point of exit at the end
15.6	The body of an iteration-statement or a selection-statement shall be a compound statement.
15.7	All `if … else if` constructs shall be terminated with an `else` statement.

Switch Statements

Rule	Description
16.1	All switch statements shall be well-formed.
16.2	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.
16.3	An unconditional break statement shall terminate every switch-clause.
16.4	Every switch statement shall have a default label.
16.5	A default label shall appear as either the first or the last switch label of a switch statement.
16.6	Every switch statement shall have at least two switch-clauses.
16.7	A switch-expression shall not have essentially Boolean type.

Functions

Rule	Description
17.1	The features of <starg.h> shall not be used.
17.3	A function shall not be declared implicitly.
17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
17.6	The declaration of an array parameter shall not contain the static keyword between the [ ].
17.7	The value returned by a function having non-void return type shall be used.

Pointers and Arrays

Rule	Description
18.4	The +, -, += and -= operators should not be applied to an expression of pointer type.
18.5	Declarations should contain no more than two levels of pointer nesting.
18.7	Flexible array members shall not be declared.
18.8	Variable-length array types shall not be used.

Overlapping Storage

Rule	Description
19.2	The `union` keyword should not be used.

Preprocessing Directives

Rule	Description
20.1	`#include` directives should only be preceded by preprocessor directives or comments.
20.2	The `'`, `"`, or `\` characters and the `/*` or `//` character sequences shall not occur in a header file name.
20.3	The `#include` directive shall be followed by either a <filename> or \"filename \" sequence.
20.4	A macro shall not be defined with the same name as a keyword.
20.5	`#undef` should not be used.
20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument.
20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.
20.8	The controlling expression of a `#if` or `#elif` preprocessing directive shall evaluate to 0 or 1.
20.9	All identifiers used in the controlling expression of `#if` or `#elif` preprocessing directives shall be `#define`'d before evaluation.
20.10	The `#` and `##` preprocessor operators should not be used.
20.11	A macro parameter immediately following a `#` operator shall not immediately be followed by a `##` operator.
20.12	A macro parameter used as an operand to the `#` or `##` operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.
20.13	A line whose first token is `#` shall be a valid preprocessing directive.
20.14	All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if`, `#ifdef` or `#ifndef` directive to which they are related.

Standard Libraries

Rule	Description
21.1	#define and #undef shall not be used on a reserved identifier or reserved macro name.
21.2	A reserved identifier or macro name shall not be declared.
21.3	The memory allocation and deallocation functions of <stdlib.h> shall not be used.
21.4	The standard header file <setjmp.h> shall not be used.
21.5	The standard header file <signal.h> shall not be used.
21.6	The Standard Library input/output functions shall not be used.
21.7	The atof, atoi, atol, and atoll functions of <stdlib.h> shall not be used.
21.8	The library functions abort, exit, getenv and system of <stdlib.h> shall not be used.
21.9	The library functions bsearch and qsort of <stdlib.h> shall not be used.
21.10	The Standard Library time and date functions shall not be used.
21.11	The standard header file <tgmath.h> shall not be used.
21.12	The exception handling features of <fenv.h> should not be used.

The rules that are checked at a system level and appear only in the system-decidable-rules subset are indicated by an asterisk.

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.

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”
Directive 4.8	Advisory	Advisory	If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden
Directive 4.12	Required	Required	Dynamic memory allocation shall not be used

Polyspace MISRA C++ Checker

The Polyspace MISRA C++ checker helps you comply with the MISRA C++:2008 coding standard.[4]

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 192 of the 228 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 11-85.

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

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

Software Quality Objective Subsets (C++)

In this section...
"SQO Subset 1 – Direct Impact on Selectivity" on page 11-85
"SQO Subset 2 – Indirect Impact on Selectivity" on page 11-87

SQO Subset 1 – Direct Impact on Selectivity

The following set of coding rules will typically improve the selectivity of your results.

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.

MISRA C++ Rule	Description
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.
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.

MISRA C++ Rule	Description
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 selectivity of your results. The following set of coding rules may help to address design issues that impact selectivity. The `SQO-subset2` option checks the rules in `SQO-subset1` and `SQO-subset2`.

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.
3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.
3-9-2	typedefs that indicate size and signedness should be used in place of the basic numerical types.
3-9-3	The underlying bit representations of floating-point values shall not be used.

<table>
<tr><th>MISRA C++ Rule</th><th>Description</th></tr>
<tr><td>4-5-1</td><td>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.</td></tr>
<tr><td>5-0-1</td><td>The value of an expression shall be the same under any order of evaluation that the standard permits.</td></tr>
<tr><td>5-0-2</td><td>Limited dependence should be placed on C++ operator precedence rules in expressions.</td></tr>
<tr><td>5-0-7</td><td>There shall be no explicit floating-integral conversions of a cvalue expression.</td></tr>
<tr><td>5-0-8</td><td>An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.</td></tr>
<tr><td>5-0-9</td><td>An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.</td></tr>
<tr><td>5-0-10</td><td>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.</td></tr>
<tr><td>5-0-13</td><td></td></tr>
<tr><td>5-0-15</td><td>Array indexing shall be the only form of pointer arithmetic.</td></tr>
<tr><td>5-0-18</td><td>>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.</td></tr>
<tr><td>5-0-19</td><td>The declaration of objects shall contain no more than two levels of pointer indirection.</td></tr>
<tr><td>5-2-1</td><td>Each operand of a logical && or || shall be a postfix - expression.</td></tr>
<tr><td>5-2-2</td><td>A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.</td></tr>
<tr><td>5-2-5</td><td>A cast shall not remove any const or volatile qualification from the type of a pointer or reference.</td></tr>
<tr><td>5-2-6</td><td>A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.</td></tr>
<tr><td>5-2-7</td><td>An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.</td></tr>
</table>

MISRA C++ Rule	Description
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.
5-2-11	The comma operator, && operator and the || operator shall not be overloaded.
5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
5-3-3	The unary & operator shall not be overloaded.
5-18-1	The comma operator shall not be used.
6-2-1	Assignment operators shall not be used in sub-expressions.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.
6-4-2	All if ... else if constructs shall be terminated with an else clause.
6-4-6	The final clause of a switch statement shall be the default-clause.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.
6-6-5	A function shall have a single point of exit at the end of the function.

MISRA C++ Rule	Description
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.
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.
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.

MISRA C++ Rule	Description
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.

MISRA C++ Coding Rules

In this section...
“Supported MISRA C++ Coding Rules” on page 11-92
“Unsupported MISRA C++ Rules” on page 11-115

Supported MISRA C++ Coding Rules

- “Language Independent Issues” on page 11-93
- “General” on page 11-93
- “Lexical Conventions” on page 11-94
- “Basic Concepts” on page 11-96
- “Standard Conversions” on page 11-97
- “Expressions” on page 11-98
- “Statements” on page 11-102
- “Declarations” on page 11-104
- “Declarators” on page 11-106
- “Classes” on page 11-107
- “Derived Classes” on page 11-107
- “Member Access Control” on page 11-108
- “Special Member Functions” on page 11-108
- “Templates” on page 11-109
- “Exception Handling” on page 11-110
- “Preprocessing Directives” on page 11-112
- “Library Introduction” on page 11-113
- “Language Support Library” on page 11-114
- “Diagnostic Library” on page 11-114
- “Input/output Library” on page 11-114

Language Independent Issues

N.	Category	MISRA Definition	Polyspace Specification
0-1-1	Required	A project shall not contain unreachable code.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
0-1-2	Required	A project shall not contain infeasible paths.	
0-1-7	Required	The value returned by a function having a non- void return type that is not an overloaded operator shall always be used.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
0-1-9	Required	There shall be no dead code.	This rule can also be enforced through detection of dead code during analysis.
0-1-10	Required	Every defined function shall be called at least once.	Detects if static functions are not called in their translation unit. Other cases are detected by the software.
0-1-11	Required	There shall be no unused parameters (named or unnamed) in nonvirtual functions.	
0-1-12	Required	There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.	Polyspace checks for unused parameters in the set of virtual functions within single translation units.
0-2-1	Required	An object shall not be assigned to an overlapping object.	

General

N.	Category	MISRA Definition	Polyspace Specification
1-0-1	Required	All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Lexical Conventions

N.	Category	MISRA Definition	Polyspace Specification
2-3-1	Required	Trigraphs shall not be used.	
2-5-1	Advisory	Digraphs should not be used.	
2-7-1	Required	The character sequence /* shall not be used within a C-style comment.	This rule cannot be annotated in the source code.
2-10-1	Required	Different identifiers shall be typographically unambiguous.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-2	Required	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.	No detection for logical scopes: fields or member functions hiding outer scopes identifiers or hiding ancestors members. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-3	Required	A typedef name (including qualification, if any) shall be a unique identifier.	No detection across namespaces. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-4	Required	A class, union or enum name (including qualification, if any) shall be a unique identifier.	No detection across namespaces. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

N.	Category	MISRA Definition	Polyspace Specification
2-10-5	Advisory	The identifier name of a non-member object or function with static storage duration should not be reused.	For functions the detection is only on the definition where there is a declaration. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-6	Required	If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.	If the identifier is a function and the function is both declared and defined then the violation is reported only once. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-13-1	Required	Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.	
2-13-2	Required	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.	
2-13-3	Required	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.	
2-13-4	Required	Literal suffixes shall be upper case.	
2-13-5	Required	Narrow and wide string literals shall not be concatenated.	

Basic Concepts

N.	Category	MISRA Definition	Polyspace Specification
3-1-1	Required	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.	
3-1-2	Required	Functions shall not be declared at block scope.	
3-1-3	Required	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.	
3-2-1	Required	All declarations of an object or function shall have compatible types.	
3-2-2	Required	The One Definition Rule shall not be violated.	Report type, template, and inline function defined in source file
3-2-3	Required	A type, object or function that is used in multiple translation units shall be declared in one and only one file.	
3-2-4	Required	An identifier with external linkage shall have exactly one definition.	
3-3-1	Required	Objects or functions with external linkage shall be declared in a header file.	
3-3-2	Required	If a function has internal linkage then all re-declarations shall include the static storage class specifier.	
3-4-1	Required	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.	

N.	Category	MISRA Definition	Polyspace Specification
3-9-1	Required	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.	Comparison is done between current declaration and last seen declaration.
3-9-2	Advisory	typedefs that indicate size and signedness should be used in place of the basic numerical types.	No detection in non-instantiated templates.
3-9-3	Required	The underlying bit representations of floating-point values shall not be used.	

Standard Conversions

N.	Category	MISRA Definition	Polyspace Specification
4-5-1	Required	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.	
4-5-2	Required	Expressions with type enum shall not be used as operands to built- in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.	
4-5-3	Required	Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator. N	

Expressions

N.	Category	MISRA Definition	Polyspace Specification
5-0-1	Required	The value of an expression shall be the same under any order of evaluation that the standard permits.	
5-0-2	Advisory	Limited dependence should be placed on C++ operator precedence rules in expressions.	
5-0-3	Required	A cvalue expression shall not be implicitly converted to a different underlying type.	Assumes that `ptrdiff_t` is signed integer
5-0-4	Required	An implicit integral conversion shall not change the signedness of the underlying type.	Assumes that `ptrdiff_t` is signed integer If the conversion is to a narrower integer with a different sign then MISRA C++ 5-0-4 takes precedence over MISRA C++ 5-0-6.
5-0-5	Required	There shall be no implicit floating-integral conversions.	This rule takes precedence over 5-0-4 and 5-0-6 if they apply at the same time.
5-0-6	Required	An implicit integral or floating-point conversion shall not reduce the size of the underlying type.	If the conversion is to a narrower integer with a different sign then MISRA C++ 5-0-4 takes precedence over MISRA C++ 5-0-6.
5-0-7	Required	There shall be no explicit floating-integral conversions of a cvalue expression.	
5-0-8	Required	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.	

N.	Category	MISRA Definition	Polyspace Specification
5-0-9	Required	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.	
5-0-10	Required	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.	
5-0-11	Required	The plain char type shall only be used for the storage and use of character values.	For numeric data, use a type which has explicit signedness.
5-0-12	Required	Signed char and unsigned char type shall only be used for the storage and use of numeric values.	
5-0-13	Required	The condition of an if-statement and the condition of an iteration-statement shall have type bool.	
5-0-14	Required	The first operand of a conditional-operator shall have type bool.	
5-0-15	Required	Array indexing shall be the only form of pointer arithmetic.	Warning on: • Operations on pointers. (p+I, I +p and p-I, where p is a pointer and I an integer, p[i] accepted). • Array indexing on nonarray pointers.
5-0-18	Required	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.	Report when relational operator are used on pointers types (casts ignored).
5-0-19	Required	The declaration of objects shall contain no more than two levels of pointer indirection.	

<table>
<tr><th>N.</th><th>Category</th><th>MISRA Definition</th><th>Polyspace Specification</th></tr>
<tr><td>5-0-20</td><td>Required</td><td>Non-constant operands to a binary bitwise operator shall have the same underlying type.</td><td></td></tr>
<tr><td>5-0-21</td><td>Required</td><td>Bitwise operators shall only be applied to operands of unsigned underlying type.</td><td></td></tr>
<tr><td>5-2-1</td><td>Required</td><td>Each operand of a logical && or || shall be a postfix - expression.</td><td>During preprocessing, violations of this rule are detected on the expressions in #if directives.
Allowed exception on associativity (a && b && c), (a || b || c).</td></tr>
<tr><td>5-2-2</td><td>Required</td><td>A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.</td><td></td></tr>
<tr><td>5-2-3</td><td>Advisory</td><td>Casts from a base class to a derived class should not be performed on polymorphic types.</td><td></td></tr>
<tr><td>5-2-4</td><td>Required</td><td>C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.</td><td></td></tr>
<tr><td>5-2-5</td><td>Required</td><td>A cast shall not remove any const or volatile qualification from the type of a pointer or reference.</td><td></td></tr>
<tr><td>5-2-6</td><td>Required</td><td>A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.</td><td>No violation if pointer types of operand and target are identical.</td></tr>
<tr><td>5-2-7</td><td>Required</td><td>An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.</td><td>"Extended to all pointer conversions including between pointer to struct object and pointer to type of the first member of the struct type. Indirect conversions through non-pointer type (e.g. int) are not detected."</td></tr>
</table>

N.	Category	MISRA Definition	Polyspace Specification
5-2-8	Required	An object with integer type or pointer to void type shall not be converted to an object with pointer type.	Exception on zero constants. Objects with pointer type include objects with pointer to function type.
5-2-9	Advisory	A cast should not convert a pointer type to an integral type.	
5-2-10	Advisory	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.	
5-2-11	Required	The comma operator, && operator and the || operator shall not be overloaded.	
5-2-12	Required	An identifier with array type passed as a function argument shall not decay to a pointer.	
5-3-1	Required	Each operand of the ! operator, the logical && or the logical || operators shall have type bool.	
5-3-2	Required	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	
5-3-3	Required	The unary & operator shall not be overloaded.	
5-3-4	Required	Evaluation of the operand to the sizeof operator shall not contain side effects.	No warning on volatile accesses and function calls
5-8-1	Required	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.	

N.	Category	MISRA Definition	Polyspace Specification
5-14-1	Required	The right hand operand of a logical && or || operator shall not contain side effects.	No warning on volatile accesses and function calls.
5-18-1	Required	The comma operator shall not be used.	
5-19-1	Required	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	

Statements

N.	Category	MISRA Definition	Polyspace Specification
6-2-1	Required	Assignment operators shall not be used in sub-expressions.	
6-2-2	Required	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.	
6-2-3	Required	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white - space character.	
6-3-1	Required	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.	
6-4-1	Required	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.	

N.	Category	MISRA Definition	Polyspace Specification
6-4-2	Required	All if ... else if constructs shall be terminated with an else clause.	Also detects cases where the last `if` is in the block of the last `else` (same behavior as JSF, stricter than MISRA C). Example: "`if` ... `else` { `if` ...}}" raises the rule
6-4-3	Required	A switch statement shall be a well-formed switch statement.	Return statements are considered as jump statements.
6-4-4	Required	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	
6-4-5	Required	An unconditional throw or break statement shall terminate every non - empty switch-clause.	
6-4-6	Required	The final clause of a switch statement shall be the default-clause.	
6-4-7	Required	The condition of a switch statement shall not have bool type.	
6-4-8	Required	Every switch statement shall have at least one case-clause.	
6-5-1	Required	A for loop shall contain a single loop-counter which shall not have floating type.	
6-5-2	Required	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	Required	The loop-counter shall not be modified within condition or statement.	Detect only direct assignments if for_index is known (see 6-5-1).

N.	Category	MISRA Definition	Polyspace Specification
6-5-4	Required	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.	
6-5-5	Required	A loop-control-variable other than the loop-counter shall not be modified within condition or expression.	
6-5-6	Required	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.	
6-6-1	Required	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	Required	The goto statement shall jump to a label declared later in the same function body.	
6-6-3	Required	The continue statement shall only be used within a well-formed for loop.	Assumes 6.5.1 to 6.5.6: so it is implemented only for supported 6_5_x rules.
6-6-4	Required	For any iteration statement there shall be no more than one break or goto statement used for loop termination.	
6-6-5	Required	A function shall have a single point of exit at the end of the function.	At most one return not necessarily as last statement for void functions.

Declarations

N.	Category	MISRA Definition	Polyspace Specification
7-3-1	Required	The global namespace shall only contain main, namespace declarations and extern "C" declarations.	

N.	Category	MISRA Definition	Polyspace Specification
7-3-2	Required	The identifier main shall not be used for a function other than the global function main.	
7-3-3	Required	There shall be no unnamed namespaces in header files.	
7-3-4	Required	using-directives shall not be used.	
7-3-5	Required	Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.	
7-3-6	Required	using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.	
7-4-2	Required	Assembler instructions shall only be introduced using the asm declaration.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
7-4-3	Required	Assembly language shall be encapsulated and isolated.	
7-5-1	Required	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.	
7-5-2	Required	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.	
7-5-3	Required	A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.	

N.	Category	MISRA Definition	Polyspace Specification
7-5-4	Advisory	Functions should not call themselves, either directly or indirectly.	

Declarators

N.	Category	MISRA Definition	Polyspace Specification
8-0-1	Required	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.	
8-3-1	Required	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.	
8-4-1	Required	Functions shall not be defined using the ellipsis notation.	
8-4-2	Required	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.	
8-4-3	Required	All exit paths from a function with non- void return type shall have an explicit return statement with an expression.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8-4-4	Required	A function identifier shall either be used to call the function or it shall be preceded by &.	
8-5-1	Required	All variables shall have a defined value before they are used.	Non-initialized variable in results and error messages for obvious cases
8-5-2	Required	Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures.	

N.	Category	MISRA Definition	Polyspace Specification
8-5-3	Required	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	

Classes

N.	Category	MISRA Definition	Polyspace Specification
9-3-1	Required	const member functions shall not return non-const pointers or references to class-data.	Class-data for a class is restricted to all non-static member data.
9-3-2	Required	Member functions shall not return non-const handles to class-data.	Class-data for a class is restricted to all non-static member data.
9-5-1	Required	Unions shall not be used.	
9-6-2	Required	Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.	
9-6-3	Required	Bit-fields shall not have enum type.	
9-6-4	Required	Named bit-fields with signed integer type shall have a length of more than one bit.	

Derived Classes

N.	Category	MISRA Definition	Polyspace Specification
10-1-1	Advisory	Classes should not be derived from virtual bases.	
10-1-2	Required	A base class shall only be declared virtual if it is used in a diamond hierarchy.	Assumes 10.1.1 not required
10-1-3	Required	An accessible base class shall not be both virtual and nonvirtual in the same hierarchy.	

N.	Category	MISRA Definition	Polyspace Specification
10-2-1	Required	All accessible entity names within a multiple inheritance hierarchy should be unique.	No detection between entities of different kinds (member functions against data members, ...).
10-3-1	Required	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.	Member functions that are virtual by inheritance are also detected.
10-3-2	Required	Each overriding virtual function shall be declared with the virtual keyword.	
10-3-3	Required	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.	

Member Access Control

N.	Category	MISRA Definition	Polyspace Specification
11-0-1	Required	Member data in non- POD class types shall be private.	

Special Member Functions

N.	Category	MISRA Definition	Polyspace Specification
12-1-1	Required	An object's dynamic type shall not be used from the body of its constructor or destructor.	
12-1-2	Advisory	All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.	
12-1-3	Required	All constructors that are callable with a single argument of fundamental type shall be declared explicit.	

N.	Category	MISRA Definition	Polyspace Specification
12-8-1	Required	A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member.	
12-8-2	Required	The copy assignment operator shall be declared protected or private in an abstract class.	

Templates

N.	Category	MISRA Definition	Polyspace Specification
14-5-2	Required	A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.	
14-5-3	Required	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.	
14-6-1	Required	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->	
14-6-2	Required	The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.	
14-7-3	Required	All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.	

N.	Category	MISRA Definition	Polyspace Specification
14-8-1	Required	Overloaded function templates shall not be explicitly specialized.	All specializations of overloaded templates are rejected even if overloading occurs after the call. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
14-8-2	Advisory	The viable function set for a function call should either contain no function specializations, or only contain function specializations.	

Exception Handling

N.	Category	MISRA Definition	Polyspace Specification
15-0-2	Advisory	An exception object should not have pointer type.	NULL not detected (see 15-1-2).
15-0-3	Required	Control shall not be transferred into a try or catch block using a goto or a switch statement.	
15-1-2	Required	NULL shall not be thrown explicitly.	
15-1-3	Required	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.	
15-3-2	Advisory	There should be at least one exception handler to catch all otherwise unhandled exceptions.	Detect that there is no try/catch in the main and that the catch does not handle all exceptions. Not detected if no "main". Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

N.	Category	MISRA Definition	Polyspace Specification
15-3-3	Required	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	Required	A class type exception shall always be caught by reference.	
15-3-6	Required	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	Required	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	Required	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	Required	A class destructor shall not exit with an exception.	Limit detection to throw and catch that are internals to the destructor; rethrows are partially processed; no detections in nested handlers.
15-5-2	Required	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).	Limit detection to throw that are internals to the function; rethrows are partially processed; no detections in nested handlers.

Preprocessing Directives

N.	Category	MISRA Definition	Polyspace Specification
16-0-1	Required	#include directives in a file shall only be preceded by other preprocessor directives or comments.	
16-0-2	Required	Macros shall only be #define 'd or #undef 'd in the global namespace.	
16-0-3	Required	#undef shall not be used.	
16-0-4	Required	Function-like macros shall not be defined.	
16-0-5	Required	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	
16-0-6	Required	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	Required	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.	
16-0-8	Required	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.	
16-1-1	Required	The defined preprocessor operator shall only be used in one of the two standard forms.	
16-1-2	Required	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	
16-2-1	Required	The preprocessor shall only be used for file inclusion and include guards.	The rule is raised for #ifdef/#define if the file is not an include file.

N.	Category	MISRA Definition	Polyspace Specification
16-2-2	Required	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.	
16-2-3	Required	Include guards shall be provided.	
16-2-4	Required	The ', ", /* or // characters shall not occur in a header file name.	
16-2-5	Advisory	The \ character should not occur in a header file name.	
16-2-6	Required	The #include directive shall be followed by either a <filename> or "filename" sequence.	
16-3-1	Required	There shall be at most one occurrence of the # or ## operators in a single macro definition.	
16-3-2	Advisory	The # and ## operators should not be used.	
16-6-1	Document	All uses of the #pragma directive shall be documented.	To check this rule, you must list the pragmas that are allowed in source files by using the option `Allowed pragmas (-allowed-pragmas)`. If Polyspace finds a pragma not in the allowed pragma list, a violation is raised.

Library Introduction

N.	Category	MISRA Definition	Polyspace Specification
17-0-1	Required	Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
17-0-2	Required	The names of standard library macros and objects shall not be reused.	

N.	Category	MISRA Definition	Polyspace Specification
17-0-5	Required	The setjmp macro and the longjmp function shall not be used.	

Language Support Library

N.	Category	MISRA Definition	Polyspace Specification
18-0-1	Required	The C library shall not be used.	
18-0-2	Required	The library functions atof, atoi and atol from library <cstdlib> shall not be used.	
18-0-3	Required	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.	The option `-compiler iso` must be used to detect violations, for example, `exit`.
18-0-4	Required	The time handling functions of library <ctime> shall not be used.	
18-0-5	Required	The unbounded functions of library <cstring> shall not be used.	
18-2-1	Required	The macro offsetof shall not be used.	
18-4-1	Required	Dynamic heap memory allocation shall not be used.	
18-7-1	Required	The signal handling facilities of <csignal> shall not be used.	

Diagnostic Library

N.	Category	MISRA Definition	Polyspace Specification
19-3-1	Required	The error indicator errno shall not be used.	

Input/output Library

N.	Category	MISRA Definition	Polyspace Specification
27-0-1	Required	The stream input/output library <cstdio> shall not be used.	

Unsupported MISRA C++ Rules

- “Language Independent Issues” on page 11-115
- “General” on page 11-116
- “Lexical Conventions” on page 11-116
- “Standard Conversions” on page 11-117
- “Expressions” on page 11-117
- “Declarations” on page 11-117
- “Classes” on page 11-118
- “Templates” on page 11-118
- “Exception Handling” on page 11-119
- “Library Introduction” on page 11-119

Language Independent Issues

N.	Category	MISRA Definition	Polyspace Specification
0–1–3	Required	A project shall not contain unused variables.	
0-1-4	Required	A project shall not contain non-volatile POD variables having only one use.	
0-1-5	Required	A project shall not contain unused type declarations.	
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.	

N.	Category	MISRA Definition	Polyspace Specification
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.
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	Polyspace Specification
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	Polyspace Specification
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.

N.	Category	MISRA Definition	Polyspace Specification
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.

Standard Conversions

N.	Category	MISRA Definition	Polyspace Specification
4-10-1	Required	ULL shall not be used as an integer value.	
4-10-2	Required	Literal zero (0) shall not be used as the null-pointer-constant.	

Expressions

N.	Category	MISRA Definition	Polyspace Specification
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.	
5-0-17	Required	Subtraction between pointers shall only be applied to pointers that 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	Polyspace Specification
7-1-1	Required	A variable which is not modified shall be const qualified.	

N.		MISRA Definition	Polyspace Specification
7-1-2	Required	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.	
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	Polyspace Specification
9-3-3	Required	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.	
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	Polyspace Specification
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.	

N.		MISRA Definition	Polyspace Specification
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	Polyspace Specification
15-0-1	Document	Exceptions shall only be used for error handling.	To observe this rule, check your compiler documentation.
15-1-1	Required	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	
15-3-1	Required	Exceptions shall be raised only after start-up and before termination of the program.	
15-3-4	Required	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	
15-5-3	Required	The terminate() function shall not be called implicitly.	

Library Introduction

N.	Category	MISRA Definition	Polyspace Specification
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.

Polyspace JSF C++ Checker

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.

5

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.

---

5. JSF and Joint Strike Fighter are Lockheed Martin.

JSF C++ Coding Rules

In this section...
"Supported JSF C++ Coding Rules" on page 11-121
"Unsupported JSF++ Rules" on page 11-145

Supported JSF C++ Coding Rules

- "Code Size and Complexity" on page 11-122
- "Environment" on page 11-122
- "Libraries" on page 11-123
- "Pre-Processing Directives" on page 11-123
- "Header Files" on page 11-125
- "Style" on page 11-125
- "Classes" on page 11-129
- "Namespaces" on page 11-133
- "Templates" on page 11-133
- "Functions" on page 11-133
- "Comments" on page 11-134
- "Declarations and Definitions" on page 11-134
- "Initialization" on page 11-135
- "Types" on page 11-136
- "Constants" on page 11-136
- "Variables" on page 11-137
- "Unions and Bit Fields" on page 11-137
- "Operators" on page 11-137
- "Pointers and References" on page 11-139
- "Type Conversions" on page 11-140
- "Flow Control Standards" on page 11-142
- "Expressions" on page 11-143
- "Memory Allocation" on page 11-144

- "Fault Handling" on page 11-144
- "Portable Code" on page 11-144

Code Size and Complexity

N.	JSF++ Definition	Polyspace Specification
1	Any one function (or method) **will** contain no more than 200 logical source lines of code (L-SLOCs).	Message in report file: *<function name>* has *<num>* logical source lines of code.
3	All functions **shall** have a cyclomatic complexity number of 20 or less.	Message in report file: *<function name>* has cyclomatic complexity number equal to *<num>*.

Environment

N.	JSF++ Definition	Polyspace Specification
8	All code **shall** conform to ISO/IEC 14882:2002(E) standard C++.	Reports the compilation error message
9	Only those characters specified in the C++ basic source character set **will** be used.	
11	Trigraphs **will not** be used.	
12	The following digraphs **will not** be used: `<%`, `%>`, `<:`, `:>`, `%:`, `%:%:`.	Message in report file: The following digraph will not be used: *<digraph>*. Reports the digraph. If the rule level is set to warning, the digraph will be allowed even if it is not supported in `-compiler iso`.
13	Multi-byte characters and wide string literals **will not** be used.	Report `L'c'`, `L"string"`, and use of `wchar_t`.
14	Literal suffixes **shall** use uppercase rather than lowercase letters.	
15	Provision **shall** be made for run-time checking (defensive programming).	Done with checks in the software.

Libraries

N.	JSF++ Definition	Polyspace Specification
17	The error indicator errno **shall not** be used.	errno should not be used as a macro or a global with external "C" linkage.
18	The macro offsetof, in library <stddef.h>, **shall not** be used.	offsetof should not be used as a macro or a global with external "C" linkage.
19	<locale.h> and the setlocale function **shall not** be used.	setlocale and localeconv should not be used as a macro or a global with external "C" linkage.
20	The setjmp macro and the longjmp function **shall not** be used.	setjmp and longjmp should not be used as a macro or a global with external "C" linkage.
21	The signal handling facilities of <signal.h> **shall not** be used.	signal and raise should not be used as a macro or a global with external "C" linkage.
22	The input/output library <stdio.h> **shall not** be used.	all standard functions of <stdio.h> should not be used as a macro or a global with external "C" linkage.
23	The library functions atof, atoi and atol from library <stdlib.h> **shall not** be used.	atof, atoi and atol should not be used as a macro or a global with external "C" linkage.
24	The library functions abort, exit, getenv and system from library <stdlib.h> **shall not** be used.	abort, exit, getenv and system should not be used as a macro or a global with external "C" linkage.
25	The time handling functions of library <time.h> **shall not** be used.	clock, difftime, mktime, asctime, ctime, gmtime, localtime and strftime should not be used as a macro or a global with external "C" linkage.

Pre-Processing Directives

N.	JSF++ Definition	Polyspace Specification
26	Only the following preprocessor directives **shall** be used: #ifndef, #define, #endif, #include.	

N.	JSF++ Definition	Polyspace Specification
27	`#ifndef`, `#define` and `#endif` **will** be used to prevent multiple inclusions of the same header file. Other techniques to prevent the multiple inclusions of header files **will not** be used.	Detects the patterns `#if !defined`, `#pragma once`, `#ifdef`, and missing `#define`.
28	The `#ifndef` and `#endif` preprocessor directives **will** only be used as defined in AV Rule 27 to prevent multiple inclusions of the same header file.	Detects any use that does not comply with AV Rule 27. Assuming 35/27 is not violated, reports only `#ifndef`.
29	The `#define` preprocessor directive **shall not** be used to create inline macros. Inline functions shall be used instead.	Rule is split into two parts: the definition of a macro function (29.def) and the call of a macrofunction (29.use). Messages in report file: • 29.1 : The `#define` preprocessor directive shall not be used to create inline macros. • 29.2 : Inline functions shall be used instead of inline macros.
30	The `#define` preprocessor directive **shall not** be used to define constant values. Instead, the `const` qualifier **shall** be applied to variable declarations to specify constant values.	Reports `#define` of simple constants.
31	The `#define` preprocessor directive **will** only be used as part of the technique to prevent multiple inclusions of the same header file.	Detects use of `#define` that are not used to guard for multiple inclusion, assuming that rules 35 and 27 are not violated.
32	The `#include` preprocessor directive **will** only be used to include header (*.h) files.	

Header Files

N.	JSF++ Definition	Polyspace Specification
33	The `#include` directive **shall** use the `<filename.h>` notation to include header files.	
35	A header file **will** contain a mechanism that prevents multiple inclusions of itself.	
39	Header files (`*.h`) **will not** contain non-const variable definitions or function definitions.	Reports definitions of global variables / function in header.

Style

N.	JSF++ Definition	Polyspace Specification
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.
43	Tabs **should** be avoided.	
44	All indentations will be at least two spaces and be consistent within the same source file.	Reports when a statement indentation is not at least two spaces more than the statement containing it. Does not report bad indentation between opening braces following if/else, do/while, for, and while statements. NB: in final release it will accept any indentation
46	User-specified identifiers (internal and external) **will not** rely on significance of more than 64 characters.	

N.	JSF++ Definition	Polyspace Specification
47	Identifiers **will not** begin with the underscore character '_'.	
48	Identifiers **will not** differ by: • Only a mixture of case • The presence/absence of the underscore character • The interchange of the letter 'O'; with the number '0' or the letter 'D' • The interchange of the letter 'I'; with the number '1' or the letter 'l' • The interchange of the letter 'S' with the number '5' • The interchange of the letter 'Z' with the number 2 • The interchange of the letter 'n' with the letter 'h'	Checked regardless of scope. Not checked between macros and other identifiers. Messages in report file: • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) only differ by the presence/absence of the underscore character. • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) only differ by a mixture of case. • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) only differ by letter `O`, with the number `0`.
50	The first word of the name of a class, structure, namespace, enumeration, or type created with `typedef` **will** begin with an uppercase letter. All others letters **will** be lowercase.	Messages in report file: • The first word of the name of a class will begin with an uppercase letter. • The first word of the namespace of a class will begin with an uppercase letter. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

N.	JSF++ Definition	Polyspace Specification
51	All letters contained in function and variables names **will** be composed entirely of lowercase letters.	Messages in report file: • All letters contained in variable names will be composed entirely of lowercase letters. • All letters contained in function names will be composed entirely of lowercase letters.
52	Identifiers for constant and enumerator values **shall** be lowercase.	Messages in report file: • Identifier for enumerator value shall be lowercase. • Identifier for template constant parameter shall be lowercase.
53	Header files **will** always have file name extension of "`.h`".	`.H` is allowed if you set the option `-dos`.
53.1	The following character sequences **shall** not appear in header file names: `'`, `\`, `/*`, `//`, or `"`.	
54	Implementation files **will** always have a file name extension of ".cpp".	Not case sensitive if you set the option `-dos`.
57	The public, protected, and private sections of a class **will** be declared in that order.	
58	When declaring and defining functions with more than two parameters, the leading parenthesis and the first argument **will** be written on the same line as the function name. Each additional argument will be written on a separate line (with the closing parenthesis directly after the last argument).	Detects that two parameters are not on the same line, The first parameter should be on the same line as function name. Does not check for the closing parenthesis.

N.	JSF++ Definition	Polyspace Specification
59	The statements forming the body of an if, else if, else, while, do ... while or for statement **shall** always be enclosed in braces, even if the braces form an empty block.	Messages in report file: • The statements forming the body of an if statement shall always be enclosed in braces. • The statements forming the body of an else statement shall always be enclosed in braces. • The statements forming the body of a while statement shall always be enclosed in braces. • The statements forming the body of a do ... while statement shall always be enclosed in braces. • The statements forming the body of a for statement shall always be enclosed in braces.
60	Braces ("`{}`") which enclose a block **will** be placed in the same column, on separate lines directly before and after the block.	Detects that statement-block braces should be in the same columns.
61	Braces ("`{}`") which enclose a block **will** have nothing else on the line except comments.	
62	The dereference operator '*' and the address-of operator '&' will be directly connected with the type-specifier.	Reports when there is a space between type and "*" "&" for variables, parameters and fields declaration.

N.	JSF++ Definition	Polyspace Specification
63	Spaces will not be used around '.' or '->', nor between unary operators and operands.	Reports when the following characters are not directly connected to a white space: • . • -> • ! • ~ • - • ++ • — **Note** A violation will be reported for "." used in float/double definition.

Classes

N.	JSF++ Definition	Polyspace Specification
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.

N.	JSF++ Definition	Polyspace Specification
75	Members of the initialization list **shall** be listed in the order in which they are declared in the class.	
76	A copy constructor and an assignment operator **shall** be declared for classes that contain pointers to data items or nontrivial destructors.	Messages in report file: • `no copy constructor and no copy assign` • `no copy constructor` • `no copy assign` 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”.

<table>
<tr><th>N.</th><th>JSF++ Definition</th><th>Polyspace Specification</th></tr>
<tr><td>81</td><td>The assignment operator shall handle self-assignment correctly</td><td>Reports when copy assignment body does not begin with “<code>if (this != arg)</code>”

A violation is not raised if an empty <code>else</code> statement follows the <code>if</code>, or the body contains only a return statement.

A violation is raised when the <code>if</code> statement is followed by a statement other than the return statement.</td></tr>
<tr><td>82</td><td>An assignment operator shall return a reference to <code>*this</code>.</td><td>The following operators should return <code>*this</code> on method, and <code>*first_arg</code> on plain function.

<code>operator=operator+=operator-=operator*=operator >>=operator <<=operator /=operator %=operator |=operator &=operator ^=prefix operator++ prefix operator--</code>

Does not report when no return exists.

No special message if type does not match.

Messages in report file:
• An assignment operator shall return a reference to <code>*this</code>.
• An assignment operator shall return a reference to its first arg.</td></tr>
<tr><td>83</td><td>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).</td><td>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.</td></tr>
</table>

N.	JSF++ Definition	Polyspace Specification
88	Multiple inheritance **shall** only be allowed in the following restricted form: `n` interfaces plus `m` private implementations, plus at most one protected implementation.	Messages in report file: • Multiple inheritance on public implementation shall not be allowed: *<public_base_class>* is not an interface. • Multiple inheritance on protected implementation shall not be allowed : *<protected_base_class_1>*. • *<protected_base_class_2>* are not interfaces.
88.1	A stateful virtual base **shall** be explicitly declared in each derived class that accesses it.	
89	A base class **shall not** be both virtual and nonvirtual in the same hierarchy.	
94	An inherited nonvirtual function **shall not** be redefined in a derived class.	Does not report for destructor. Message in report file: Inherited nonvirtual function `%s` shall not be redefined in a derived class.
95	An inherited default parameter **shall never** be redefined.	
96	Arrays **shall not** be treated polymorphically.	Reports pointer arithmetic and array like access on expressions whose pointed type is used as a base class.
97	Arrays **shall not** be used in interface.	Only to prevent array-to-pointer-decay. Not checked on private methods
97.1	Neither operand of an equality operator (== or `!=`) **shall** be a pointer to a virtual member function.	Reports == and `!=` on pointer to member function of polymorphic classes (cannot determine statically if it is virtual or not), except when one argument is the null constant.

Namespaces

N.	JSF++ Definition	Polyspace Specification
98	Every nonlocal name, except `main()`, **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 Specification
104	A template specialization **shall** be declared before its use.	Reports the actual compilation error message.

Functions

N.	JSF++ Definition	Polyspace Specification
107	Functions **shall** always be declared at file scope.	
108	Functions with variable numbers of arguments **shall not** be used.	
109	A function definition should not be placed in a class specification unless the function is intended to be inlined.	Reports when "inline" is not in the definition of a member function inside the class definition.
110	Functions with more than 7 arguments **will not** be used.	
111	A function **shall not** return a pointer or reference to a non-static local object.	Simple cases without alias effect detected.
113	Functions **will** have a single exit point.	Reports first return, or once per function.
114	All exit points of value-returning functions **shall** be through return statements.	
116	Small, concrete-type arguments (two or three words in size) **should** be passed by value if changes made to formal parameters should not be reflected in the calling function.	Report constant parameters references with `sizeof <= 2 * sizeof(int)`. Does not report for copy-constructor.

N.	JSF++ Definition	Polyspace Specification
119	Functions **shall** not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).	Direct recursion is reported statically. Indirect recursion reported through the software. Message in report file: Function <*F*> shall not call directly itself.
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 Specification
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).	Reports when a file does not begin with two comment lines. **Note**: This rule cannot be annotated in the source code.

Declarations and Definitions

N.	JSF++ Definition	Polyspace Specification
135	Identifiers in an inner scope **shall not** use the same name as an identifier in an outer scope, and therefore hide that identifier.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

N.	JSF++ Definition	Polyspace Specification
136	Declarations should be at the smallest feasible scope.	Reports when: • A global variable is used in only one function. • A local variable is not used in a statement (`expr`, `return`, `init` ...) of the same level of its declaration (in the same block) or is not used in two sub-statements of its declaration. **Note** • Non-used variables are reported. • Initializations at definition are ignored (not considered an access)
137	All declarations at file scope should be static where possible.	
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 Specification
142	All variables **shall** be initialized before use.	Done with Non-initialized variable checks in the software.

N.	JSF++ Definition	Polyspace Specification
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 Specification
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 Specification
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.

N.	JSF++ Definition	Polyspace Specification
151.1	A string literal shall not be modified.	Report when a `char*`, `char[]`, or `string` type is used not as `const`. A violation is raised if a string literal (for example, “ “) is cast as a non `const`.

Variables

N.	JSF++ Definition	Polyspace Specification
152	Multiple variable declarations **shall not** be allowed on the same line.	

Unions and Bit Fields

N.	JSF++ Definition	Polyspace Specification
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 Specification
157	The right hand operand of a `&&` or `||` operator shall not contain side effects.	Assumes rule 159 is not violated. Messages in report file: • The right hand operand of a `&&` operator shall not contain side effects. • The right hand operand of a `||` operator shall not contain side effects.

<table>
<tr><th>N.</th><th>JSF++ Definition</th><th>Polyspace Specification</th></tr>
<tr><td>158</td><td>The operands of a logical && or || shall be parenthesized if the operands contain binary operators.</td><td>Messages in report file:
• The operands of a logical && shall be parenthesized if the operands contain binary operators.
• The operands of a logical || shall be parenthesized if the operands contain binary operators.
Exception for: X || Y || Z , Z&&Y &&Z</td></tr>
<tr><td>159</td><td>Operators ||, &&, and unary & shall not be overloaded.</td><td>Messages in report file:
• Unary operator & shall not be overloaded.
• Operator || shall not be overloaded.
• Operator && shall not be overloaded.</td></tr>
<tr><td>160</td><td>An assignment expression shall be used only as the expression in an expression statement.</td><td>Only simple assignment, not +=, ++, etc.</td></tr>
<tr><td>162</td><td>Signed and unsigned values shall not be mixed in arithmetic or comparison operations.</td><td></td></tr>
<tr><td>163</td><td>Unsigned arithmetic shall not be used.</td><td></td></tr>
<tr><td>164</td><td>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).</td><td></td></tr>
<tr><td>164.1</td><td>The left-hand operand of a right-shift operator shall not have a negative value.</td><td>Detects constant case +. Found by the software for dynamic cases.</td></tr>
<tr><td>165</td><td>The unary minus operator shall not be applied to an unsigned expression.</td><td></td></tr>
<tr><td>166</td><td>The sizeof operator will not be used on expressions that contain side effects.</td><td></td></tr>
<tr><td>168</td><td>The comma operator shall not be used.</td><td></td></tr>
</table>

Pointers and References

N.	JSF++ Definition	Polyspace Specification
169	Pointers to pointers should be avoided when possible.	Reports second-level pointers, except for arguments of main.
170	More than 2 levels of pointer indirection **shall not** be used.	Only reports on variables/parameters.
171	Relational operators shall not be applied to pointer types except where both operands are of the same type and point to: • the same object, • the same function, • members of the same object, or • elements of the same array (including one past the end of the same array).	Reports when relational operator are used on pointer types (casts ignored).
173	The address of an object with automatic storage **shall not** be assigned to an object which persists after the object has ceased to exist.	
174	The null pointer **shall not** be de-referenced.	Done with checks in software.
175	A pointer **shall not** be compared to `NULL` or be assigned `NULL`; use plain `0` instead.	Reports usage of `NULL` macro in pointer contexts.
176	A `typedef` **will** be used to simplify program syntax when declaring function pointers.	Reports non-`typedef` function pointers, or pointers to member functions for types of variables, fields, parameters. Returns type of function, cast, and exception specification.

Type Conversions

N.	JSF++ Definition	Polyspace Specification
177	User-defined conversion functions **should** be avoided.	Reports user defined conversion function, non-explicit constructor with one parameter or default value for others (even undefined ones). Does not report copy-constructor. Additional message for constructor case: This constructor should be flagged as "explicit".
178	Down casting (casting from base to derived class) **shall** only be allowed through one of the following mechanism: • Virtual functions that act like dynamic casts (most likely useful in relatively simple cases). • Use of the visitor (or similar) pattern (most likely useful in complicated cases).	Reports explicit down casting, dynamic_cast included. (Visitor patter does not have a special case.)
179	A pointer to a virtual base class **shall not** be converted to a pointer to a derived class.	Reports this specific down cast. Allows dynamic_cast.

N.	JSF++ Definition	Polyspace Specification

180	Implicit conversions that may result in a loss of information **shall not** be used.	Reports the following implicit casts :
		
		`integer => smaller integer`
		`unsigned => smaller or eq signed`
		`signed => smaller or eq un-signed`
		`integer => float float => integer`
		
		Does not report for cast to `bool` reports for implicit cast on constant done with the options `-scalar-overflows-checks signed-and-unsigned` or `-ignore-constant-overflows`
		
		Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

181	Redundant explicit casts **will not** be used.	Reports useless cast: `cast T to T`. Casts to equivalent `typedefs` are also reported.
182	Type casting from any type to or from pointers **shall not** be used.	Does not report when Rule 181 applies.
184	Floating point numbers **shall not** be converted to integers unless such a conversion is a specified algorithmic requirement or is necessary for a hardware interface.	Reports `float->int` conversions. Does not report implicit ones.
185	C++ style casts (`const_cast`, `reinterpret_cast`, and `static_cast`) **shall** be used instead of the traditional C-style casts.	

Flow Control Standards

N.	JSF++ Definition	Polyspace Specification
186	There **shall** be no unreachable code.	Done with gray checks in the software. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
187	All non-null statements **shall** potentially have a side-effect.	
188	Labels **will not** be used, except in switch statements.	
189	The `goto` statement **shall** not be used.	
190	The `continue` statement **shall not** be used.	
191	The `break` statement **shall not** be used (except to terminate the cases of a switch statement).	
192	All `if`, `else if` constructs will contain either a final `else` clause or a comment indicating why a final `else` clause is not necessary.	`else if` should contain an `else` clause.
193	Every non-empty `case` clause in a switch statement **shall** be terminated with a `break` statement.	
194	All `switch` statements that do not intend to test for every enumeration value **shall** contain a final `default` clause.	Reports only for missing `default`.
195	A `switch` expression **will** not represent a Boolean value.	
196	Every `switch` statement **will** have at least two cases and a potential `default`.	
197	Floating point variables **shall not** be used as loop counters.	Assumes 1 loop parameter.

N.	JSF++ Definition	Polyspace Specification
198	The initialization expression in a `for` loop **will** perform no actions other than to initialize the value of a single `for` loop parameter.	Reports if `loop` parameter cannot be determined. Assumes Rule 200 is not violated. The `loop variable` parameter is assumed to be a variable.
199	The increment expression in a `for` loop **will** perform no action other than to change a single loop parameter to the next value for the loop.	Assumes 1 loop parameter (Rule 198), with non class type. Rule 200 must not be violated for this rule to be reported.
200	Null initialize or increment expressions in `for` loops **will not** be used; a `while` loop will be used instead.	
201	Numeric variables being used within a *for* loop for iteration counting shall not be modified in the body of the loop.	Assumes 1 loop parameter (AV rule 198), and no alias writes.

Expressions

N.	JSF++ Definition	Polyspace Specification
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: • by itself • the right-hand side of an assignment • a condition • the only argument expression with a side-effect in a function call • condition of a loop • switch condition • single part of a chained operation	Reports when: • A side effect is found in a return statement • A side effect exists on a single value, and only one operand of the function call has a side effect.

N.	JSF++ Definition	Polyspace Specification
204.1	The value of an expression shall be the same under any order of evaluation that the standard permits.	Reports when: • 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 Specification
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 Specification
208	C++ exceptions **shall not** be used.	Reports `try`, `catch`, `throw spec`, and `throw`.

Portable Code

N.	JSF++ Definition	Polyspace Specification
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`.

N.	JSF++ Definition	Polyspace Specification
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 11-146
- "Rules" on page 11-146
- "Environment" on page 11-146
- "Libraries" on page 11-147
- "Header Files" on page 11-147
- "Style" on page 11-147
- "Classes" on page 11-147
- "Namespaces" on page 11-149
- "Templates" on page 11-149
- "Functions" on page 11-150
- "Comments" on page 11-150
- "Initialization" on page 11-151
- "Types" on page 11-151
- "Unions and Bit Fields" on page 11-151
- "Operators" on page 11-151
- "Type Conversions" on page 11-151
- "Expressions" on page 11-152
- "Memory Allocation" on page 11-152

- "Portable Code" on page 11-152
- "Efficiency Considerations" on page 11-152
- "Miscellaneous" on page 11-153
- "Testing" on page 11-153

Code Size and Complexity

N.	JSF++ Definition
2	There shall not be any self-modifying code.

Rules

N.	JSF++ Definition
4	To break a "should" rule, the following approval must be received by the developer: • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool)
5	To break a "will" or a "shall" rule, the following approvals must be received by the developer: • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool) • approval from the software product manager (obtained by the unit approval in the developmental CM tool)
6	Each deviation from a "shall" rule shall be documented in the file that contains the deviation. Deviations from this rule shall not be allowed, AV Rule 5 notwithstanding.
7	Approval will not be required for a deviation from a "shall" or "will" rule that complies with an exception specified by that rule.

Environment

N.	JSF++ Definition
10	Values of character types will be restricted to a defined and documented subset of ISO 10646 1.

Libraries

N.	JSF++ Definition
16	Only DO-178B level A [15] certifiable or SEAL 1 C/C++ libraries shall be used with safety-critical (i.e. SEAL 1) code.

Header Files

N.	JSF++ Definition
34	Header files should contain logically related declarations only.
36	Compilation dependencies should be minimized when possible.
37	Header (include) files should include only those header files that are required for them to successfully compile. Files that are only used by the associated .cpp file should be placed in the .cpp file — not the .h file.
38	Declarations of classes that are only accessed via pointers (*) or references (&) should be supplied by forward headers that contain only forward declarations.

Style

N.	JSF++ Definition
45	All words in an identifier will be separated by the '_' character.
49	All acronyms in an identifier will be composed of uppercase letters.
55	The name of a header file should reflect the logical entity for which it provides declarations.
56	The name of an implementation file should reflect the logical entity for which it provides definitions and have a ".cpp" extension (this name will normally be identical to the header file that provides the corresponding declarations.) At times, more than one .cpp file for a given logical entity will be required. In these cases, a suffix should be appended to reflect a logical differentiation.

Classes

N.	JSF++ Definition
64	A class interface should be complete and minimal.
65	A structure should be used to model an entity that does not require an invariant.
66	A class should be used to model an entity that maintains an invariant.

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	The invariant for a class should be: • A part of the postcondition of every class constructor, • A part of the precondition of the class destructor (if any), • A part of the precondition and postcondition of every other publicly accessible operation.
73	Unnecessary default constructors shall not be defined.
77	A copy constructor shall copy all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).
80	The default copy and assignment operators will be used for classes when those operators offer reasonable semantics.
84	Operator overloading will be used sparingly and in a conventional manner.
85	When two operators are opposites (such as == and !=), both will be defined and one will be defined in terms of the other.
86	Concrete types should be used to represent simple independent concepts.
87	Hierarchies should be based on abstract classes.
90	Heavily used interfaces should be minimal, general and abstract.
91	Public inheritance will be used to implement “is-a” relationships.

N.	JSF++ Definition
92	A subtype (publicly derived classes) will conform to the following guidelines with respect to all classes involved in the polymorphic assignment of different subclass instances to the same variable or parameter during the execution of the system: • Preconditions of derived methods must be at least as weak as the preconditions of the methods they override. • Postconditions of derived methods must be at least as strong as the postconditions of the methods they override. In other words, subclass methods must expect less and deliver more than the base class methods they override. This rule implies that subtypes will conform to the Liskov Substitution Principle.
93	"has-a" or "is-implemented-in-terms-of" relationships will be modeled through membership or non-public inheritance.

Namespaces

N.	JSF++ Definition
100	Elements from a namespace should be selected as follows: • using declaration or explicit qualification for few (approximately five) names, • using directive for many names.

Templates

N.	JSF++ Definition
101	Templates shall be reviewed as follows: 1 with respect to the template in isolation considering assumptions or requirements placed on its arguments. 2 with respect to all functions instantiated by actual arguments.
102	Template tests shall be created to cover all actual template instantiations.
103	Constraint checks should be applied to template arguments.
105	A template definition's dependence on its instantiation contexts should be minimized.
106	Specializations for pointer types should be made where appropriate.

Functions

N.	JSF++ Definition
112	Function return values should not obscure resource ownership.
115	If a function returns error information, then that error information will be tested.
117	Arguments should be passed by reference if NULL values are not possible: • **117.1** – An object should be passed as `const T&` if the function should not change the value of the object. • **117.2** – An object should be passed as `T&` if the function may change the value of the object.
118	Arguments should be passed via pointers if NULL values are possible: • **118.1** – An object should be passed as `const T*` if its value should not be modified. • **118.2** – An object should be passed as `T*` if its value may be modified.
120	Overloaded operations or methods should form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal parameters.
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.

N.	JSF++ Definition
131	One should avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).
132	Each variable declaration, typedef, enumeration value, and structure member will be commented.
134	Assumptions (limitations) made by functions should be documented in the function’s preamble.

Initialization

N.	JSF++ Definition
143	Variables will not be introduced until they can be initialized with meaningful values. (See also AV Rule 136, AV Rule 142, and AV Rule 73 concerning declaration scope, initialization before use, and default constructors respectively.)

Types

N.	JSF++ Definition
146	Floating point implementations shall comply with a defined floating point standard. The standard that will be used is the ANSI/IEEE® Std 754 [1].

Unions and Bit Fields

N.	JSF++ Definition
155	Bit-fields will not be used to pack data into a word for the sole purpose of saving space.

Operators

N.	JSF++ Definition
167	The implementation of integer division in the chosen compiler shall be determined, documented and taken into account.

Type Conversions

N.	JSF++ Definition
183	Every possible measure should be taken to avoid type casting.

Expressions

N.	JSF++ Definition
204	A single operation with side-effects shall only be used in the following contexts: 1 by itself 2 the right-hand side of an assignment 3 a condition 4 the only argument expression with a side-effect in a function call 5 condition of a loop 6 switch condition 7 single part of a chained operation

Memory Allocation

N.	JSF++ Definition
207	Unencapsulated global data will be avoided.

Portable Code

N.	JSF++ Definition
210	Algorithms shall not make assumptions concerning how data is represented in memory (e.g. big endian vs. little endian, base class subobject ordering in derived classes, nonstatic data member ordering across access specifiers, etc.).
210.1	Algorithms shall not make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier.
211	Algorithms shall not assume that shorts, ints, longs, floats, doubles or long doubles begin at particular addresses.
212	Underflow or overflow functioning shall not be depended on in any special way.
214	Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.

Efficiency Considerations

N.	JSF++ Definition
216	Programmers should not attempt to prematurely optimize code.

Miscellaneous

N.	JSF++ Definition
217	Compile-time and link-time errors should be preferred over run-time errors.
218	Compiler warning levels will be set in compliance with project policies.

Testing

N.	JSF++ Definition
219	All tests applied to a base class interface shall be applied to all derived class interfaces as well. If the derived class poses stronger postconditions/invariants, then the new postconditions /invariants shall be substituted in the derived class tests.
220	Structural coverage algorithms shall be applied against flattened classes.
221	Structural coverage of a class within an inheritance hierarchy containing virtual functions shall include testing every possible resolution for each set of identical polymorphic references.

12

Checking Coding Rules

- "Set Up Coding Rules Checking" on page 12-2
- "Create Custom Coding Rules" on page 12-6
- "Format of Custom Coding Rules File" on page 12-8
- "Exclude Files from Rules Checking" on page 12-10
- "Configure Additional Options for Certain Rules" on page 12-11
- "Review Coding Rule Violations" on page 12-13
- "Filter and Group Coding Rule Violations" on page 12-15
- "Generate Coding Rules Report" on page 12-18
- "Coding Rules Not Checked in Compilation Phase" on page 12-20

Set Up Coding Rules Checking

You can use Polyspace to look for coding rule violations. You can look for violation of:

- MISRA or JSF coding rules.

  For more information, see “Coding Rules”.
- Naming conventions for identifiers that you define.

  For more information, see “Custom Coding Rules”.

With the exception of certain rules on page 12-20, Polyspace checks for coding rule violations during the compilation phase. If you want to check for coding rules only, you can run Polyspace on your source code upto this phase. For information on the analysis option that controls upto which phase you can run, see `Verification level (-to)`.

Tip Polyspace Code Prover does not support checking of the following:

- MISRA C:2012 Directive 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

For support of all MISRA C: 2012 rules, use Polyspace Bug Finder.

Select Predefined Coding Rule Sets

You can check for violations of predefined subsets of MISRA or JSF coding rules.

User Interface	Command Line
Select your project configuration. On the **Configuration** pane, select **Coding Rules & Code Metrics**. Select an appropriate option. For more information on the options, see “Coding Rules & Code Metrics”.	Use the appropriate option with the `polyspace-code-prover-nodesktop` command or `polyspaceCodeProver` function. For more information on the options, see the section **Command-Line Information** in “Coding Rules & Code Metrics”.

Select Specific MISRA or JSF Coding Rules

You can check for violations of specific MISRA or JSF coding rules.

User Interface	Command Line

1 Select your project configuration. On the **Configuration** pane, select **Coding Rules & Code Metrics**.
2 Select the option corresponding to the coding rules whose violations you want to check: MISRA C: 2004, MISRA C: 2012, MISRA C++, etc.
3 From the dropdown list for the option, select `custom`.
 	
The software displays a new field for your custom file.
4 To the right of this field, click **Edit**. A New File window opens, displaying a table of rules.
 	
Select the check box for the rules that you want to check.
5 Click **OK** to save the rules and close the window.
 	
The full path to the rules file appears. To reuse this rules file for other projects, type this path name or use the icon in the New File window.
	1 Create a coding rules file in one of the two ways:
	• Create the file from the user interface.
	• Enter the rules directly in a text file.
	
	On each line, specify a coding rule in the format:
	
	`Rule_number on|off #Comments`
	
	For example:
	
	`10.5 off # rule 10.5: essential type model`
	`17.2 on # rule 17.2: functions`
	
	You can only enter the rules that you want to turn off. When you run an analysis, Polyspace automatically turns on the other rules and populates the file.
	2 Use the appropriate option with the `polyspace-code-prover-nodesktop` command or `polyspaceCodeProver` function.
	
	Provide the full path to the file you created as option argument. For example:
	
	`polyspace-code-prover-nodesktop -misra3 C:\myRulesFile.txt`

Create Coding Rules

You can create your own coding rules to enforce naming conventions for identifiers in your source code. 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.

User Interface	Command Line

1 Select your project configuration. On the **Configuration** pane, select **Coding Rules & Code Metrics**.
2 Select the **Check custom rules** box.
3 Click Edit.
The New File window opens, displaying a table of rule groups.
4 Clear the **Custom rules** check box to turn off checking of custom rules.
5 For each rule you want to turn on:
a Select the corresponding check box.
b Specify the naming convention and associated error message if the convention is violated.
For more information, see `Check custom rules (-custom-rules)`.
 	
For a tutorial with a specific code, see “Create Custom Coding Rules” on page 12-6.
	1 Create a coding rules file in one of the two ways:
	• Create the file from the user interface.
	• Enter the rules directly in a text file.
	On each line, specify a coding rule in the format:
	`*Rule_number* on|off #*Comments*`
	`convention=*violation_message*`
	`pattern=*regular_expression*`
	For example:
	`8.1  on  # Global constants`
	`convention=Global constants`
	`  must begin by G_ and`
	`  must be in capital letters.`
	`pattern=G_[A-Z0-9_]`
	2 Use the option `-custom-rules` with the `polyspace-code-prover-nodesktop` command or `polyspaceCodeProver` function.
	Provide the full path to the file you created as option argument. For example:
	`polyspace-code-prover-nodesktop -custom-rules C:\myRulesFile.txt`

See Also

Related Examples

- "Exclude Files from Rules Checking" on page 12-10
- "Configure Additional Options for Certain Rules" on page 12-11
- "Review Coding Rule Violations" on page 12-13

Create Custom Coding Rules

This tutorial shows how to create a custom coding rules file. You can use this file to check names or text patterns in your source code with reference to 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.

For the high-level workflow, see “Create Coding Rules” on page 12-4.

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);
}
```

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

   The New File window opens, displaying a table of rule groups.
4 Specify the rules to check for.

   a First, clear the **Custom rules** check box to turn off checking of custom rules.

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

Column Title	Action
Status	Select ☑.

Column Title	Action
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 you had entered, `All struct fields must begin with s_ and have capital letters`

6 Right-click on 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.

See Also

Polyspace Analysis Options

`Check custom rules (-custom-rules)`

Related Examples

- “Exclude Files from Rules Checking” on page 12-10

More About

- “Rule Checking” on page 11-2
- “Format of Custom Coding Rules File” on page 12-8

Format of Custom Coding Rules File

In a custom coding rules file, each rule appears in the following format:

```
N.n off|on
convention=violation_message
pattern=regular_expression
```

- *N.n* — Custom rule number, for example, 1.2.
- `off` — Rule is not considered.
- `on` — The software checks for violation of the rule. After analysis, it displays the coding rule violation on the **Results List** pane.
- *violation_message* — Software displays this text in an XML file within the *Results*/`Polyspace-Doc` folder.
- *regular_expression* — Software compares this text pattern against a source code identifier that is specific to the rule. See "Custom Coding Rules".

  The keywords `convention=` and `pattern=` are optional. If present, they apply to the rule whose number immediately precedes these keywords. If `convention=` is not given for a rule, then a standard message is used. If `pattern=` is not given for a rule, then the default regular expression is used, that is, `.*`.

  Use the symbol `#` to start a comment. Comments are not allowed on lines with the keywords `convention=` and `pattern=`.

  The following example contains three custom rules: 1.1, 8.1, and 9.1.

```
# Custom rules configuration file
1.1  off          # Disable custom rule number 1.1
8.1  on       # Violation of custom rule 8.1 produces a warning
convention=Global constants must begin by G_ and must be in capital letters.
pattern=G_[A-Z0-9_]*
9.1  on    # Non-adherence to custom rule 9.1 produces a warning
convention=Global variables should begin by g_.
pattern=g_.*
```

See Also

Related Examples

- "Create Custom Coding Rules" on page 12-6

Exclude Files from Rules Checking

This example shows how to specify files that you do not want analyzed for coding rule violations. For instance, sometimes, you have to add header files from a third-party library to your Polyspace project for a precise analysis, but you cannot address rule violations in those header files. Therefore, you do not want coding rule violations on those files.

By default:

- Results are generated for all source files and header files in the same folders as source files.
- Results are not generated for the remaining header files in your project.

You can change this default behavior and specify your own set of files on which you do not want coding rule violations. You can suppress coding rule violations and code metrics, but not run-time checks.

Use a combination of the following options to suppress results from files in which you are not interested. In the Polyspace user interface, the options appear on the **Configuration** pane under the **Inputs & Stubbing** node.

- `Generate results for sources and (-generate-results-for)`
- `Do not generate results for (-do-not-generate-results-for)`

For instance, you can suppress results from certain folders and unsuppress them only for certain files in those folders.

See Also

Related Examples

- “Set Up Coding Rules Checking” on page 12-2

More About

- “Rule Checking” on page 11-2

Configure Additional Options for Certain Rules

To check certain MISRA C rules, you must provide additional information outside your code.

Specify Allowed Custom Pragma Directives

This example shows how to exclude custom pragma directives from coding rules checking. MISRA C:2004 rule 3.4 requires checking that all pragma directives are documented within the documentation of the compiler. However, you can allow undocumented pragma directives to be present in your code.

User Interface	Command Line
1 Select your project configuration. On the **Configuration** pane, select **Coding Rules & Code Metrics**. 2 Select **Check MISRA C:2004** or **Check MISRA AC AGC**. The **Allowed pragmas** option appears. 3 Enter the allowed pragma names. For more information, see `Allowed pragmas (-allowed-pragmas)`.	Use the option `-allowed-pragmas` with the `polyspace-code-prover-nodesktop` command or `polyspaceCodeProver` function. For more information, see the section **Command-Line Information** in `Allowed pragmas (-allowed-pragmas)`.

Specify Effective Boolean Types

This example shows how to specify data types you want Polyspace to consider as Boolean during MISRA C rules checking. The software applies this redefinition only to data types defined by `typedef` statements.

The use of this option is related to checking of the following rules:

- MISRA C:2004 and MISRA AC AGC — 12.6, 13.2, 15.4.

  For more information, see “MISRA C:2004 Rules”.
- MISRA C:2012 — 10.1, 10.3, 10.5, 14.4 and 16.7.

User Interface	Command Line
1 Select your project configuration. On the **Configuration** pane, select **Coding Rules & Code Metrics**. 2 Select one of the predefined coding rule options. The **Effective boolean types** option appears. 3 Enter the type names. For more information, see `Effective boolean types (-boolean-types)`.	Use the option `-boolean-types` with the `polyspace-code-prover-nodesktop` command or `polyspaceCodeProver` function. For more information, see the section **Command-Line Information** in `Effective boolean types (-boolean-types)`.

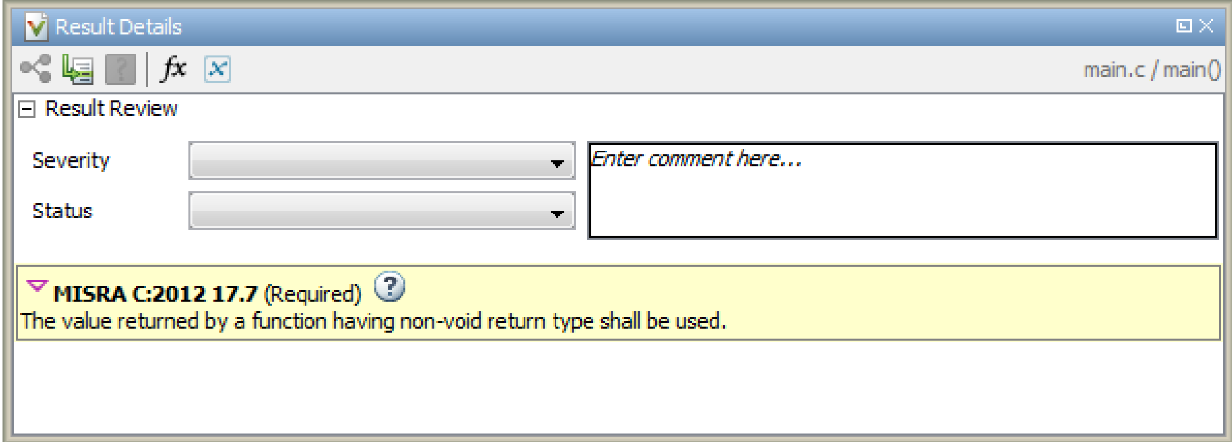
Review Coding Rule Violations

This example shows how to review coding rule violations in the Polyspace user interface once code analysis is complete. After analysis, the **Results List** pane displays the rule violations.

1 Select a coding-rule violation on the **Results List** pane.

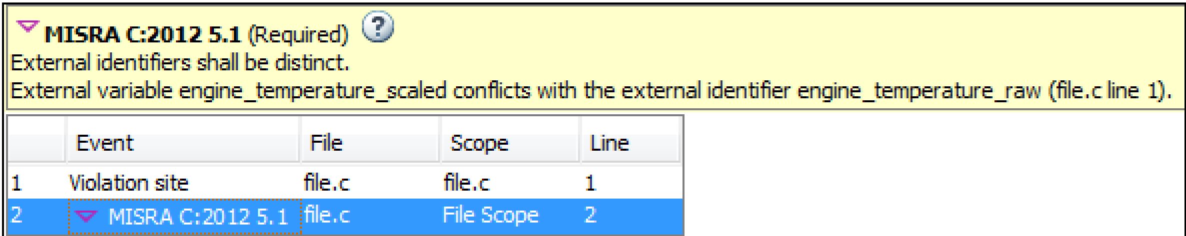
- The predefined rules such as MISRA or JSF are indicated by ▽ .
- The custom rules are indicated by ▼ .

2 On the **Result Details** pane, view the location and description of the violated rule. In the source code, the line containing the violation appears highlighted.



3 For certain rules, use additional information available for investigating the rule violation.

- For MISRA C: 2012 rules, on the **Result Details** pane, click the ? icon to see the rationale for the rule. In some cases, you can also see code examples illustrating the violation.
- For MISRA C: 2012 and MISRA C++ rules that involve more than one location in the code, the **Result Details** pane shows both locations as an event history. To navigate to a location in the source code, click the corresponding event.



4 Review the violation in your code. Determine whether you must fix the code to avoid the violation.

- If you decide to fix the code, to open the source file that contains the coding rule violation, on the **Source** pane, right-click the code with the purple check. Select **Open Editor**. The file opens in the **Code Editor** pane or an external text editor depending on your **Preferences**.
- Otherwise, add a comment and justification in your result or code explaining why you did not change the code. For more information, see "Add Review Comments to Results" on page 8-32 or "Justify Results Through Code Annotations" on page 8-36.

See Also

Related Examples

- "Set Up Coding Rules Checking" on page 12-2
- "Filter and Group Coding Rule Violations" on page 12-15

Filter and Group Coding Rule Violations

This example shows how to use filters in the **Results List** pane to focus on specific kinds of coding rule violations. By default, the **Results List** pane displays all coding rule violations and run-time checks.

Filter Coding Rules

1. On the **Results List** pane, select the icon on the **Check** column header.
2. From the context menu, clear the **All** check box.
3. Select the violated rule numbers that you want to focus on.
4. Click **OK**.

To filter out all results other than coding rule violations, use the filters on the **Type** or **Family** column header.

You can also filter rule violations using the **Top 5 coding rule violations** graph on the **Dashboard** pane. See “Filter and Group Results” on page 8-113.

Group Coding Rules

1. On the **Results List** pane, from the list, select **Family**.

   The rules are grouped by numbers. Each group corresponds to a certain code construct.
2. Expand the group nodes to select an individual coding rule violation.

Suppress Certain Rules from Display in One Click

Instead of filtering individual rules from display each time you open your results, you can limit the display of rule violations in one click. Use the drop-down list in the left of the **Results List** pane toolbar. You can add some predefined options to this 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 violations of certain coding rules.

1. Select **Tools** > **Preferences**.

2 On the **Review Scope** tab, do one of the following:

- To add predefined options to the drop-down list on the **Results List** pane, select **Include Quality Objectives Scopes**.

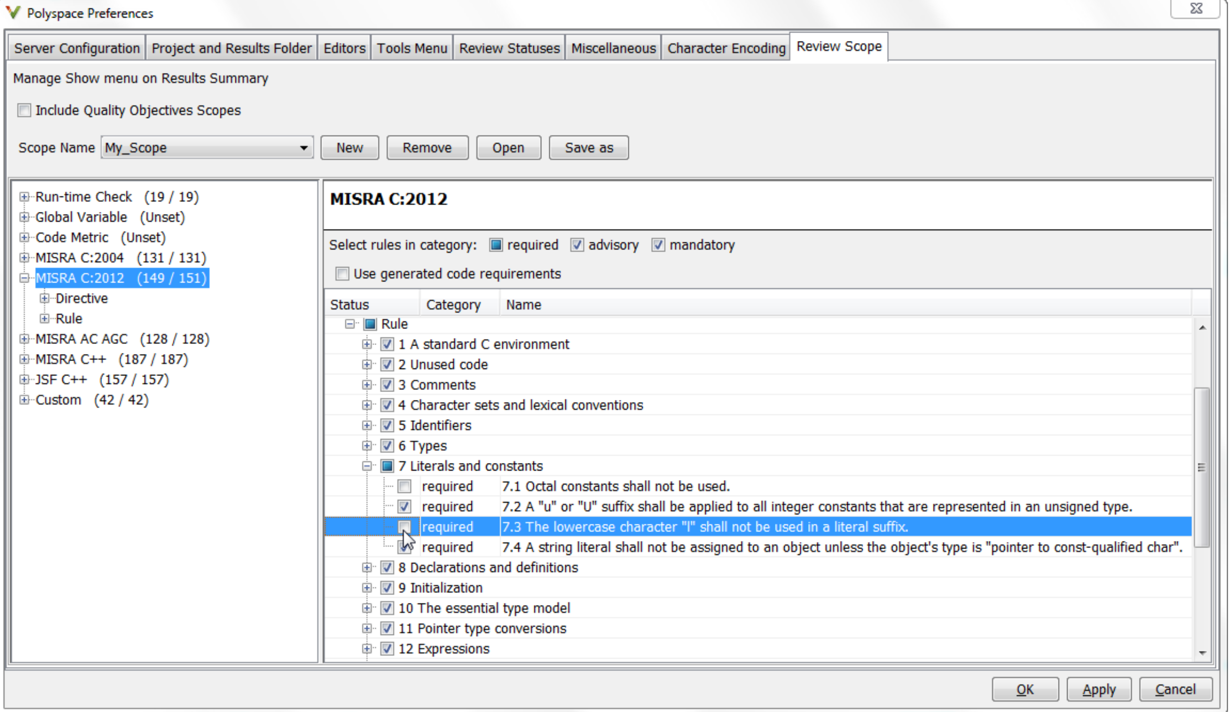
  The **Scope Name** list shows additional options, `HIS`, `SQO-4`, `SQO-5`, and `SQO-6`. Select an option to see which rules are suppressed from display.

  In addition to coding rule violations, the options impose limits on the display of code metrics and orange checks. For a detailed description of the predefined options, see "Software Quality Objectives" on page 8-122.
- To create your own option in the drop-down list on the **Results List** pane, select **New**. Save your option file.

  On the left pane, select a rule set such as **MISRA C:2012**. On the right pane, to suppress a rule from display, clear the box next to the rule.

  To suppress all rules belonging to a group, such as **The essential type model**, clear the box next to the group name. For more information on the groups, see "Coding Rules". If you select only a fraction of rules in a group, the check box next to the group name displays a ■ symbol.

  To suppress all rules belonging to a category, such as **advisory**, clear the box next to the category name on the top of the right pane. If you select only a fraction of rules in a category, the check box next to the category name displays a ■ symbol.



3 Click **Apply** or **OK**.

On the **Results List** pane, the drop-down list on the left displays the additional options.

4 Select the option that you want. The rules that you suppress do not appear on the **Results List** pane.

See Also

Related Examples

- “Set Up Coding Rules Checking” on page 12-2
- “Review Coding Rule Violations” on page 12-13
- “Filter and Group Results” on page 8-113

Generate Coding Rules Report

This example shows how to generate and view a coding rules report after verification.

Generate Report

1. With the results open, select **Reporting** > **Run Report**.
2. In the Run Report dialog box, from the **Select Reports** menu, select `CodingRules`.
3. Specify **Output folder** and **Output format**.
4. Select **Run Report**.

Open Existing Report

1. With the results open, select **Reporting** > **Open Report**.
2. In the Open Report dialog box, navigate to the folder that contains the coding rules report.

   The default location is *`ResultFolder`*`\Polyspace-Doc`.
3. Select the report and click **OK**.

View Report

In the coding rules report, you can view the following information:

- **Summary for all Files** — Lists number of violations in each file.
- **Summary for Enabled Rules** — For each rule, lists the:
  - Rule number.
  - Rule description.
  - Number of times the rule is broken.
- **Violations** — For each file that Polyspace checked for coding rule violations, lists each violation along with the:
  - Rule description.
  - Unique ID for the violation. Use this ID to find the violation on the **Results List** pane.
  - Function where the rule violation is found.

  - Line and column number.
  - Review information you enter such as **Class**, **Status** and **Comment**.
- **Configuration Settings** — Lists analysis options used for the verification, along with coding rules that Polyspace checked.

See Also

Related Examples

- "Set Up Coding Rules Checking" on page 12-2
- "Customize Existing Report Template" on page 8-144

Coding Rules Not Checked in Compilation Phase

The following coding rules are not checked entirely during compilation phase:

- MISRA C:2004 — 9.1, 13.7 and 21.1

  For more information, see "MISRA C:2004 Rules".
- MISRA C:2012

  - `MISRA C:2012 Rule 2.2`
  - `MISRA C:2012 Rule 9.1`
  - `MISRA C:2012 Rule 14.3`
  - `MISRA C:2012 Rule 18.1`

To check for violations of these rules, you must run your analysis past the compilation phase.

For all other rules, if you want to detect rule violations alone, you can run only the compilation phase of the analysis. For more information, see "Set Up Coding Rules Checking" on page 12-2.

Software Quality with Polyspace Metrics

- “Code Quality Metrics” on page 13-2
- “Generate Code Quality Metrics” on page 13-12
- “View Code Quality Metrics” on page 13-19
- “Compare Metrics Against Software Quality Objectives” on page 13-23
- “View Trends in Code Quality Metrics” on page 13-29
- “Web Browser Requirements for Polyspace Metrics” on page 13-32
- “Elements in Custom Software Quality Objectives File” on page 13-33

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 Rules”.
- 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 13-23.

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 13-21.

Column Name		Description
Verification Status		Verification level completed. See `Verification level (-to)`.
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 "Add Review Comments to Results" on page 8-32 or "Justify Results Through Code Annotations" on page 8-36.
	Violations	Total number of coding rule violations.
Run-Time Errors	**Confirmed Defects**	Number of run-time checks to which you assign a **Severity** of `High`, `Medium` or `Low` in the Polyspace user interface. See "Add Review Comments to Results" on page 8-32 or "Justify Results Through Code Annotations" on page 8-36.
	Run-Time Reliability	A measure of your code quality, expressed as a percentage. The percentage is calculated as number of green and other justified checks divided by the total number of checks. To justify a check, in the Polyspace user interface, you must assign an appropriate **Status**. See "Add Review Comments to Results" on page 8-32 or "Justify Results Through Code Annotations" on page 8-36.

Column Name		Description
Software Quality Objectives	**Overall Status**	A status of **PASS** or **FAIL** based on whether your code satisfies the software quality objectives you specified. For more information, see "Compare Metrics Against Software Quality Objectives" on page 13-23.
	Level	The software quality objectives that you specify. You can either use a predefined set of objectives or specify your own objectives. See: • "Software Quality Objectives" on page 8-122 • "Customize Software Quality Objectives" on page 13-25
	Review Progress	A measure of your review progress, expressed as a percentage. 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. To review a check, in the Polyspace user interface, you must assign a **Status**. See "Add Review Comments to Results" on page 8-32 or "Justify Results Through Code Annotations" on page 8-36.

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 "Add Review Comments to Results" on page 8-32 or "Justify Results Through Code Annotations" on page 8-36.
	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 "Add Review Comments to Results" on page 8-32 or "Justify Results Through Code Annotations" on page 8-36.
	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 "Add Review Comments to Results" on page 8-32 or "Justify Results Through Code Annotations" on page 8-36.

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 Rules”.

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 “Add Review Comments to Results” on page 8-32 or “Justify Results Through Code Annotations” on page 8-36.
	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 “Add Review Comments to Results” on page 8-32 or “Justify Results Through Code Annotations” on page 8-36.
	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 13-23.

Column Name		Description
	Level	The software quality objectives that you specify. You can either use a predefined set of objectives, or specify your own objectives. See: • "Software Quality Objectives" on page 8-122 • "Customize Software Quality Objectives" on page 13-25
	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 "Add Review Comments to Results" on page 8-32 or "Justify Results Through Code Annotations" on page 8-36.

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 "Add Review Comments to Results" on page 8-32 or "Justify Results Through Code Annotations" on page 8-36.

<table>
<tr><th colspan="2">Column Name</th><th>Description</th></tr>
<tr><td colspan="2">Run-Time Selectivity</td><td>Percentage, calculated as the number of non-orange checks divided by the total number of checks.</td></tr>
<tr><td>Green Code</td><td>Checks</td><td>Number of green checks.

See “Result and Source Code Colors” on page 8-3.</td></tr>
<tr><td rowspan="2">Systematic Run-Time Errors (Red Checks)</td><td>Justified</td><td>Percentage of red checks that you have justified.

To justify a check, in the Polyspace user interface, assign an appropriate Status. See “Add Review Comments to Results” on page 8-32 or “Justify Results Through Code Annotations” on page 8-36.</td></tr>
<tr><td>Checks</td><td>Number of red checks.

See “Result and Source Code Colors” on page 8-3.</td></tr>
<tr><td rowspan="2">Unreachable Branches (Gray Checks)</td><td>Justified</td><td>Percentage of gray checks that you have justified.

To justify a check, in the Polyspace user interface, assign an appropriate Status. See “Add Review Comments to Results” on page 8-32 or “Justify Results Through Code Annotations” on page 8-36.</td></tr>
<tr><td>Checks</td><td>Number of gray checks.

See “Result and Source Code Colors” on page 8-3.</td></tr>
</table>

Column Name		Description
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 “Add Review Comments to Results” on page 8-32 or “Justify Results Through Code Annotations” on page 8-36.
	Checks	Number of orange checks. See “Result and Source Code Colors” on page 8-3.
	Path-Related Issues	Number of orange checks that indicate a run-time error only on certain execution paths. See “Critical Orange Checks” on page 10-13.
	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 10-13.
	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 10-13.
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“Add Review Comments to Results” on page 8-32 or “Justify Results Through Code Annotations” on page 8-36.

Column Name		Description
	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 13-23.
	Level	The software quality objectives that you specify. You can either use a predefined set of objectives or specify your own objectives. See: • “Software Quality Objectives” on page 8-122 • “Customize Software Quality Objectives” on page 13-25
	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 “Add Review Comments to Results” on page 8-32 or “Justify Results Through Code Annotations” on page 8-36.

See Also

Related Examples

- “Generate Code Quality Metrics” on page 13-12
- “View Code Quality Metrics” on page 13-19
- “Compare Metrics Against Software Quality Objectives” on page 13-23
- “View Trends in Code Quality Metrics” on page 13-29

Generate Code Quality Metrics

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 13-2.

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

Upload Results to Polyspace Metrics

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

- If you perform a remote verification:

  1. On the **Configuration** pane, select **Run Settings**.
  2. 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.
  3. If you do not select **Upload results to Polyspace Metrics**, after verification, the results are downloaded to your computer.

     To upload them later to the Polyspace Metrics web interface, select **Metrics > Upload to Metrics**.
  4. 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 MJS host are always encrypted. To use a secure web data transfer with HTTPS, see "Configure Web Server for HTTPS".

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

- If you perform a local verification, to upload your results to the Polyspace Metrics web interface, select **Metrics** > **Upload to Metrics**.

  Command Line: Use the command `polyspace-results-repository`. For information on the command, use the `-h` flag. At the command line, enter:

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

  Here, *`matlabroot`* is the MATLAB installation folder, for instance, `C:\Program Files\MATLAB`.

For more information, see "View Code Quality Metrics" on page 13-19.

Specify Automatic Uploading of Results

Note This functionality will be removed in a future release.

You can:

- Configure verifications to start automatically and periodically, for example, at a specific time every night.
- Specify that Polyspace must upload your results automatically to the Polyspace Metrics web interface.
- Specify that Polyspace will send you an email after uploading the results. This email contains:
  - Links to results
  - If the verification produces compilation errors, an attached log file

- A summary of new findings, for example, new coding rule violations, and new potential and actual run-time errors

To configure automatic verification and uploading of results:

1 Save the following content in an XML file. Name the file `Projects.psproj`.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Polyspace Metrics Automatic Verification Project File -->
<Configuration>
<Project name="Demo_C" language="C" verificationKind="INTEGRATION"
product="CODE-PROVER">
  <Options>
    -O2
    -to pass2
    -target sparc
    -entry-points tregulate,proc1,proc2
    -critical-section-begin Begin_CS:CS1
    -critical-section-end End_CS:CS1
    -misra2 all-rules
    -do-not-generate-results-for sources/math.h
    -D NEW_DEFECT
  </Options>
  <LaunchingPeriod hour="12" minute="20" month="*" weeDay="1">
  </LaunchingPeriod>
  <Commands>
    <GetSource>
      /bin/cp -vr /yourcompany/home/auser/tempfolder/Demo_C_Studio/sources/ .
    </GetSource>
    <GetVersion>
    </GetVersion>
  </Commands>
  <Users>
    <User>
      <FirstName>Polyspace</FirstName>
      <LastName>User</LastName>
      <Mail resultsMail="ALWAYS"
    compilationFailureMail="yes">userid@yourcompany.com</Mail>
    </User>
  </Users>
</Project>
<SmtpConfiguration server="smtp.yourcompany.com" port="25">
</SmtpConfiguration>
</Configuration>
```

2 Save this file in the results repository on the Polyspace Metrics server. For example:

`/var/Polyspace/results-repository`

3 Modify the contents of this file appropriately.

Replace	Replace with
`Demo_C`	Project name.

Replace	Replace with
`C`	`C` or `CPP`: Source code language.
`INTEGRATION`	`INTEGRATION` or `UNIT-BY-UNIT`: Verification mode. For more information on `UNIT-BY-UNIT`, see "Run File-by-File Remote Verification" on page 6-14.
`CODE-PROVER`	`BUG-FINDER` or `CODE-PROVER`: Product name.

<Options>
    -O2
    -to pass2
    -target sparc
    -temporal-exclusions-file sources/^
temporal_exclusions.txt
    -entry-points tregulate,proc1,proc2,^
server1,server2
    -critical-section-begin Begin_CS:CS1
    -critical-section-end End_CS:CS1
    -misra2 all-rules
    -do-not-generate-results-for sources/^
math.h
    -D NEW_DEFECT
  </Options>
	Your own set of options in <Options>..</Options>. For more information, see "Analysis Options".

`12`	Integer in 0–23: Starting hour of verification.
`20`	Integer in 0–59: Starting minute of verification.
`*`	One of the following: • Integer in 1–12: Starting month of verification. • `*` to specify once every month. • *n1*-*n2* to specify a range. For example, `1-5`.

Replace	Replace with
`1`	One of the following: • Integer in 1–7: Starting weekday of verification. • `*` to specify once every day. • *`n1`*`-`*`n2`* to specify a range. For example, `1-5`.
`/bin/cp -vr /yourcompany/home/auser/tempfolder/Demo_C_Studio/sources/`	Optional command to retrieve source files from the configuration management system, or the file system of the user. The command is executed in a temporary folder on the client computer, which is also used to store results from the compilation phase of the verification. This temporary folder is removed after the verification is moved to the Polyspace server.
`Polyspace`	Your first name.
`User`	Your last name.

Replace	Replace with
ALWAYS	One of the following: • ALWAYS: An email is always sent at the end of each automatic verification • NEW-CERTAIN-FINDINGS: An email is sent only if there are new red or gray checks. • NEW-POTENTIAL-FINDINGS: An email is sent only if there are new orange checks. • NEW-CODING-RULES-FINDINGS: An email is sent only if there are new coding rule violations. • ALL-NEW-FINDINGS: An email is sent if there are new checks or coding-rule violations.
yes	yes or no: An email is sent if verification fails because of compilation failure.
user_id@yourcompany.com	Your email address.
smtp.yourcompany.com	Your Simple Mail Transport Protocol (SMTP) server.
25	Your SMTP server port.

See Also

Related Examples

- “View Code Quality Metrics” on page 13-19
- “Compare Metrics Against Software Quality Objectives” on page 13-23

- "View Trends in Code Quality Metrics" on page 13-29

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 13-12.

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.

Goal	Action
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 13-21.

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 13-21.
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 13-23.

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 13-2.

2 On the **Code Metrics**, **Coding Rules** or **Run-Time Errors** tabs, select an entry to download the result to the Polyspace user interface.

Note 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.

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

3 In the Polyspace user interface, review the particular result, investigate the root cause in your source code, and assign review comments and justifications.

4 To upload your comments and justifications to the Polyspace Metrics repository, select **Metrics** > **Upload to Metrics**.

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

a Select **Tools** > **Preferences**.

b On the **Server Configuration** tab, select **Save justifications in the Polyspace Metrics repository**.

---

5 After your review is over, in the Polyspace Metrics interface, click Refresh to view updated metrics.

See Also

Related Examples

- "View Trends in Code Quality Metrics" on page 13-29

Compare Metrics Against Software Quality Objectives

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 13-19.

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 13-19.
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 8-122.
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 `SQO-1`, but do not check for coding rule violations in your project, Polyspace cannot determine whether your project satisfies all the objectives specified in `SQO-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 13-21.

6 After your review, in the Polyspace Metrics interface, click Refresh 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. See "Import Review Comments from Previous Verifications" on page 8-33.

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>
```

2 Save this XML file in the folder where remote analysis data is stored, for example, `C:\Users\JohnDoe\AppData\Roaming\Polyspace_RLDatas`.

   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 13-33.

   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.

Rule	String	Rule numbers
MISRA C: 2004	`MISRA_C_`	“MISRA C:2004 and MISRA AC AGC Coding Rules” on page 11-14
MISRA C: 2012	`MISRA_C3_`	“MISRA C:2012 Directives and Rules”
MISRA C++	`MISRA_Cpp_`	“MISRA C++ Coding Rules” on page 11-92
JSF C++	`JSF_Cpp_`	“JSF C++ Coding Rules” on page 11-121
Custom coding rules	`Custom_`	“Custom Coding Rules” (Polyspace Bug Finder)

c For specifying checks, use the appropriate check acronym. For more information, see “Acronyms for Checks and Code Metrics” on page 8-69.

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-SQO-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 SQO level 'Custom-SQO-Level' is multiply defined.
  ```
- `ParentID` specifies another level from which the current level inherits its quality objectives. In the preceding example, the level `Custom-SQO-Level_2` inherits its quality objectives from the level `Custom-SQO-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-SQO-Level_2`.

  The quality thresholds that you specify override the thresholds that `Custom-SQO-Level_2` inherits from `Custom-SQO-Level`. For instance, if you specify `<goto>1</goto>`, this overrides the threshold specification `<goto>0</goto>` of `Custom-SQO-Level`.

See Also

Related Examples

- “View Trends in Code Quality Metrics” on page 13-29

View Trends in Code Quality Metrics

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 13-19.

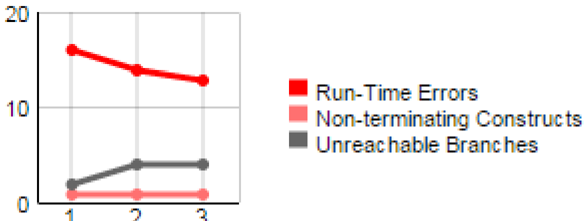
2 On the **Projects** tab, select the project for which you want to view trends.

   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 ▲	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 `Not a defect`.

See Also

Related Examples

- "Code Quality Metrics" on page 13-2

Web Browser Requirements for Polyspace Metrics

Polyspace Metrics supports the following 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

To use Polyspace Metrics, install Java, version 1.4 or later on your computer.

For the Firefox web browser, manually install the required Java plug-in. For example, if your computer uses the Linux operating system:

1. Create a Firefox folder for plug-ins:

```
mkdir ~/.mozilla/plugins
```

2. Go to this folder:

```
cd ~/.mozilla/plugins
```

3. Create a symbolic link to the Java plug-in, which is available in the Java Runtime Environment folder of your MATLAB installation:

```
ln -s MATLAB_Install/sys/java/jre/glnxa64/jre/lib/amd64/libnpjp2.so
```

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 13-25.

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

Element	Metric
`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`
`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 Unprotected Shared Variables`

Configure Model for Code Analysis

- “Configure Simulink Model” on page 14-2
- “Recommended Model Settings for Code Analysis” on page 14-3
- “Check Simulink Model Settings” on page 14-6
- “Annotate Blocks for Known Results” on page 14-12

Configure Simulink Model

Before analyzing your generated code, there are certain settings that you should apply to your model. Use the following workflow to prepare your model for code analysis.

- If you know of results ahead of time, annotate your blocks on page 14-12 with Polyspace annotations.
- Set the recommended configuration parameters on page 14-3.
- Double-check your model settings on page 14-6.
- Generate code.
- Set up your Polyspace options.

Recommended Model Settings for Code Analysis

For Polyspace analyses, set the following parameter configurations before generating code. If you do not use the recommended value for `SystemTargetFile`, you get an error. For other parameters, if you do not use the recommended value, you get a warning.

Grouping	Command-Line	Name and Location in Configuration
Code Generation	Name: `SystemTargetFile (Simulink Coder)` Value: An Embedded Coder Target Language Compiler (TLC) file. For example `ert.tlc` or `autosar.tlc`.	Location: **Code Generation** Name: **System target file** Value: Embedded Coder target file
	Name: `MatFileLogging (Simulink Coder)` Value: `'off'`	Location: **Code Generation > Interface** Name: **MAT-file logging** Value: ☐ Not selected
	Name: `GenerateReport (Simulink Coder)` Value: `'on'`	Location: **Code Generation > Report** Name: **Create code-generation report** Value: ☑ Selected
	Name: `IncludeHyperlinksInReport (Simulink Coder)` Value: `'on'`	Location: **Code Generation > Report** Name: **Code-to-model** Value: ☑ Selected

Grouping	Command-Line	Name and Location in Configuration
	Name: `GenerateSampleERTMain (Embedded Coder)` Value: `'off'`	Location: **Code Generation > Templates** Name: **Generate an example main program** Value: ☐ Not selected
	Name: `GenerateComments (Simulink Coder)` Value: `'on'`	Location: **Code Generation > Comments** Name: **Include comments** Value: ☑ Selected
Optimization	Name: `DefaultParameterBehavior (Simulink)` Value: `'Inlined'`	Location: **Optimization** > **Signals and Parameters** Name: **Default parameter behavior** Value: `Inlined`
	Name: `InitFltsAndDblsToZero (Simulink)` Value: `'on'`	Location: **Optimization** Name: **Use memset to initialize floats and doubles to 0.0** Value: ☐ Not selected
	Name: `ZeroExternalMemoryAtStartup (Simulink)` Value: `'on'`	Location: **Optimization** Name: **Remove root level I/O zero initialization** Value: ☐ Not selected

Grouping	Command-Line	Name and Location in Configuration
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)`

Check Simulink Model Settings

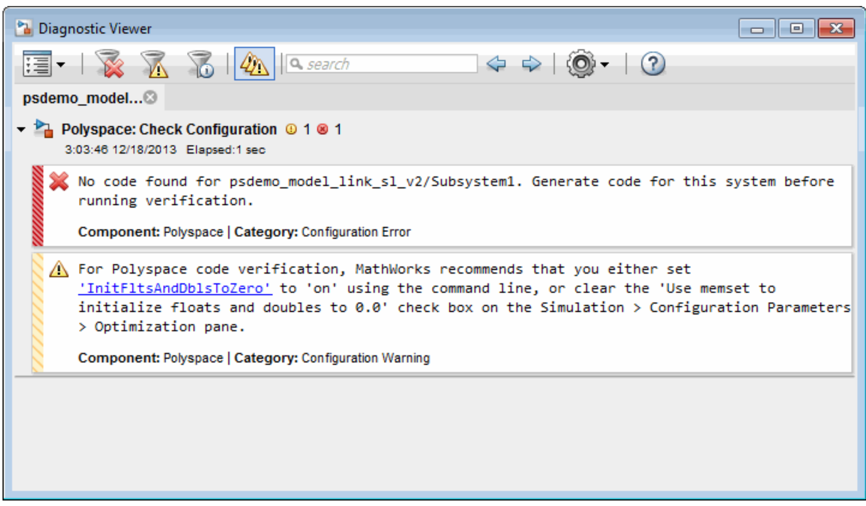
With the Polyspace plug-in, you can check your model settings before generating code or before starting an analysis. If you alter your model settings, rebuild the model to generate fresh code. If the generated code version does not match your model version, warnings appear when you run the analysis.

Check Simulink Model Settings Using the Code Generation Advisor

Before generating code, you can check your model settings against the "Recommended Model Settings for Code Analysis" on page 14-3. If you do not use the recommended model settings, the back-to-model linking will not work correctly.

1. From the Simulink model window, select **Code** > **C/C++ Code** > **Code Generation Options**. The Configuration Parameters dialog box opens, displaying the **Code Generation** pane.
2. Select **Set Objectives**.
3. From the **Set Objective – Code Generation Advisor** window, add the `Polyspace` objective and any others that you want to check.
4. In the **Check model before generating code** drop-down list, select either:
   - `On (stop for warnings)`, the process stops for either errors or warnings without generating code.
   - `On (proceed with warnings)`, the process stops for errors, but continues generating code if the configuration only has warnings.
5. Select **Check Model**.

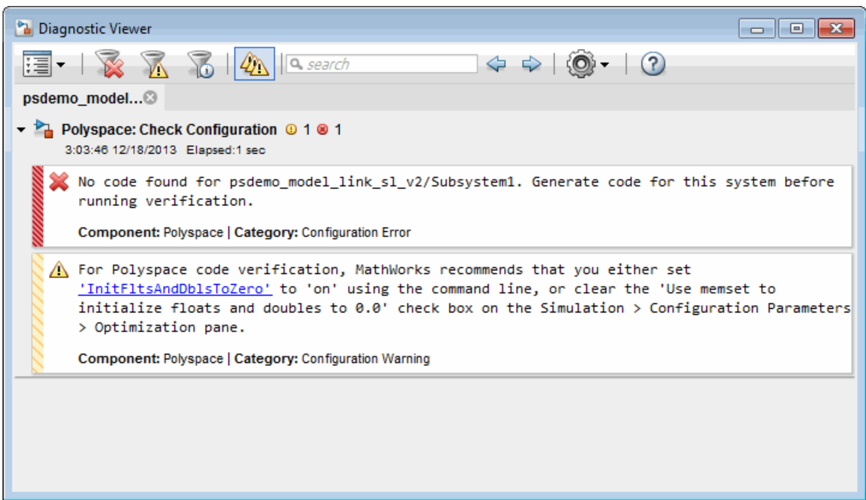
   The software runs a configuration check. If your configuration check finds errors or warnings, the **Diagnostics Viewer** displays the issues and recommendations.



Check Simulink Model Settings Before Analysis

With the Polyspace plug-in, you can check your model settings before starting an analysis:

1. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.
2. Click **Check configuration**. If your model settings are not optimal for Polyspace, the software displays warning messages with recommendations.

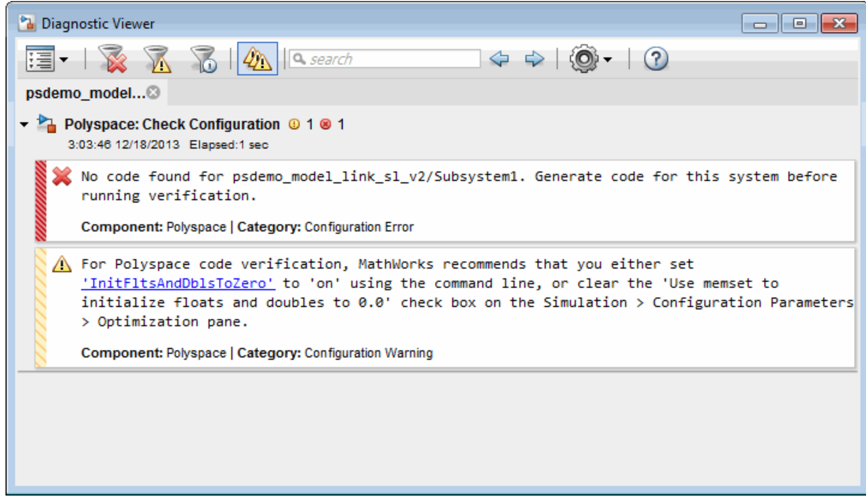


3 From the **Check configuration before verification** menu, select either:

- `On (stop for warnings)`, the analysis stops for either errors or warnings.
- `On (proceed with warnings)`, the analysis stops for errors, but continues the code analysis if the configuration only has warnings.

4 Select **Run verification**.

The software runs a configuration check. If your configuration check finds errors or warnings, the **Diagnostics Viewer** displays the issues and recommendations.

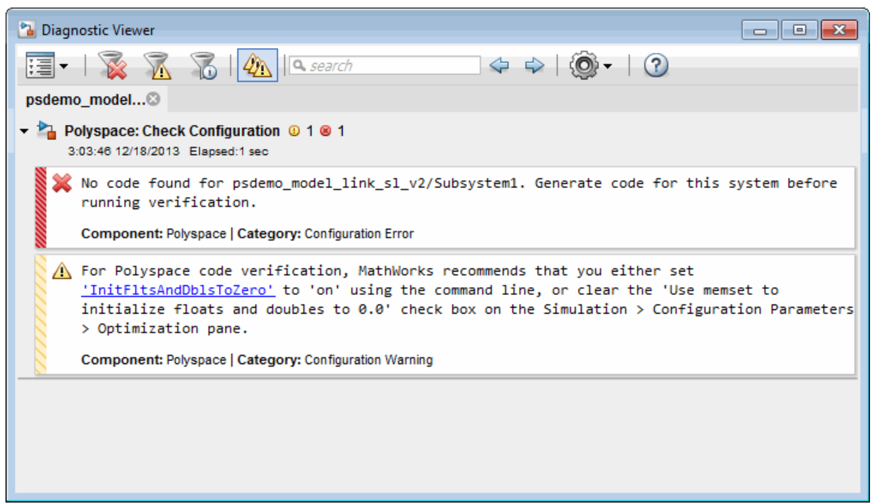


If you alter your model settings, rebuild the model to generate fresh code. If the generated code version does not match your model version, the software produces warnings when you run the analysis.

Check Simulink Model Settings Automatically

With the Polyspace plug-in, you can check your model settings before starting an analysis:

1. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.
2. Click **Check configuration**. If your model settings are not optimal for Polyspace, the software displays warning messages with recommendations.

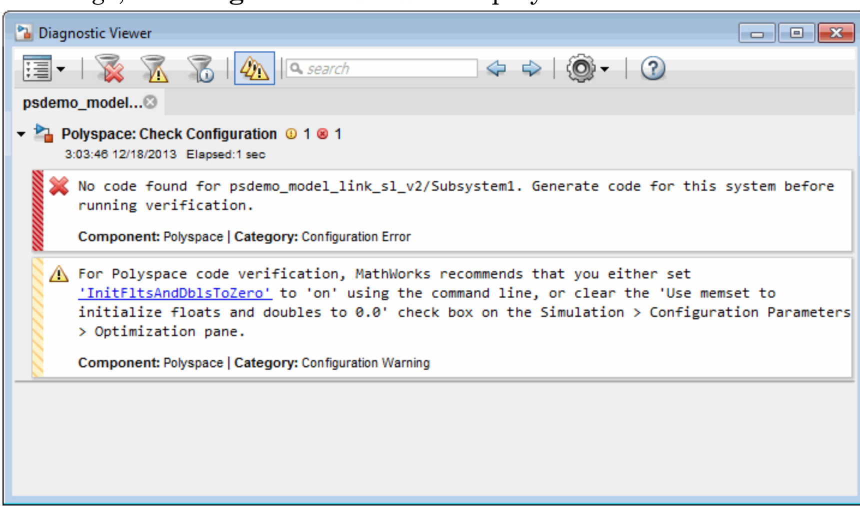


3. From the **Check configuration before verification** menu, select either:

   - `On (stop for warnings)` — will
   - `On (proceed with warnings)`

4. Select **Run verification**.

   The software runs a configuration check. If your configuration check finds errors or warnings, the **Diagnostics Viewer** displays the issues and recommendations.



If you select:

- `On (stop for warnings)`, the analysis stops for either errors or warnings.
- `On (proceed with warnings)` — the analysis stops for errors, but continues the code analysis if the configuration only has warnings.

If you alter your model settings, rebuild the model to generate fresh code. If the generated code version does not match your model version, the software produces warnings when you run the analysis.

See Also

More About

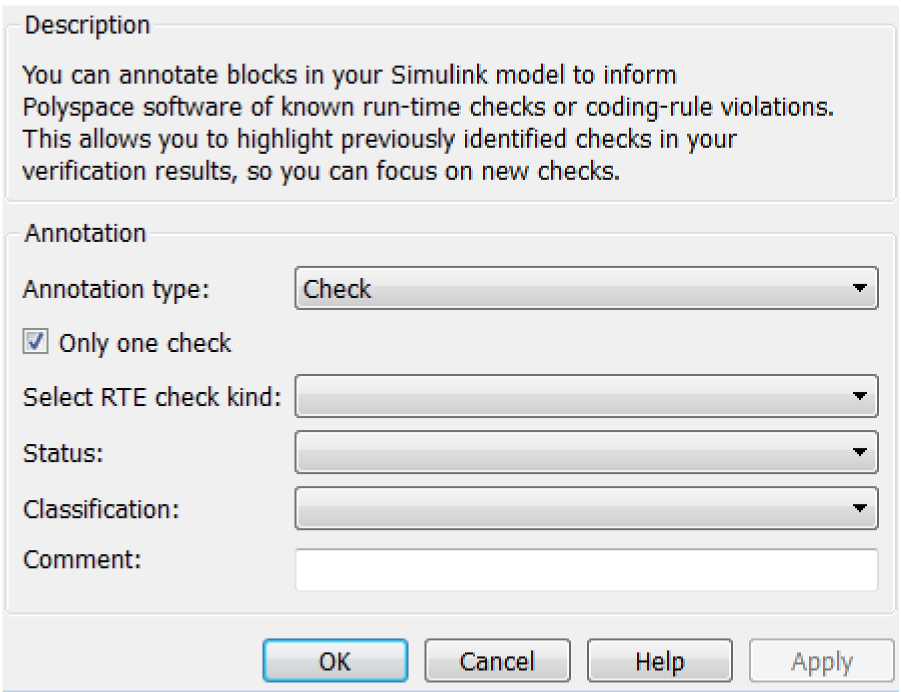
- "Recommended Model Settings for Code Analysis" on page 14-3

Annotate Blocks for Known Results

You can annotate individual blocks in your Simulink model to inform Polyspace software of known defects, run-time checks, or coding-rule violations. These annotations allow you to highlight and categorize previously identified results, so you can focus on reviewing new results.

Your Polyspace results displays the information that you provide with block annotations. To annotate blocks:

1 Right-click the block you want to annotate and select **Polyspace** > **Annotate Selected Block** > **Edit**.



2 In the Polyspace Annotation dialog box, select an **Annotation type**:

- `Check` — Code Prover run-time error
- `Defect` — Bug Finder defect
- `MISRA-C` — MISRA C 2004 coding rule violation

- `MISRA-AC-AGC` — MISRA AC AGC coding rule violation
- `MISRA-C-2012` — MISRA C 2012 coding rule violation
- `MISRA-C++` — MISRA C++ coding rule violation
- `JSF` — JSF C++ coding rule violation

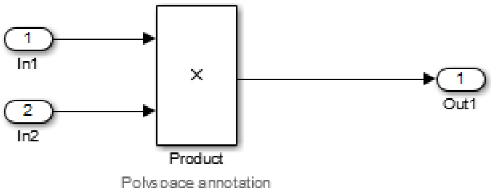
3 If you want to highlight only one kind of result, select **Only 1 check** and the relevant error or coding rule from the **Select *result* kind** drop-down list.

4 If you want to highlight a list of checks, clear **Only 1 check**. In the **Enter a list of *results*** field, specify the errors or rules that you want to highlight.

5 Set any of the following options as desired:

Option	Values
Status – describe how you intend to address the issue	`To investigate`, `To fix`, `Justified`, `No action planned`, `Other` The last two statuses mark the result as justified.
Severity — describe the severity of the issue	`High`, `Medium`, `Low`, `Not a defect`
Comment	Any additional information about the check.

6 Click **OK**. Your annotation is added to the block.



When you run an analysis, the **Results List** pane pre-populates the results with your annotation.

Family	Check	Information	File	Classification	Status	Comment
✕	Unused variable	Variable: errno	__polyspace__stdstubs.c			
?	Out of bounds array index	Origin: Possibly impacted by inputs	controller.c			
?	Division by Zero	Origin: Possibly impacted by inputs	controller.c	Medium	Improve	Remove zero
?	Overflow	Origin: Possibly impacted by inputs	command_strategy_file.c			
?	Overflow	Origin: Possibly impacted by inputs	controller.c			
?	Overflow	Origin: Possibly impacted by inputs	controller.c			

See Also

`pslinkfun`

15

Polyspace Code Prover Analysis in Simulink

- “Install Polyspace Plugin for Simulink” on page 15-2
- “Verify Generated Code Using Polyspace Code Prover” on page 15-4
- “Verify Code Generated from Simulink Subsystem” on page 15-7
- “Auto-Annotate Generated Code to Justify Checks” on page 15-16
- “Main Generation for Model Verification” on page 15-18
- “Configure Data Range Settings” on page 15-20
- “Embedded Coder Considerations” on page 15-22
- “TargetLink Considerations” on page 15-27
- “View Results in Polyspace Code Prover” on page 15-30
- “Identify Errors in Simulink Models” on page 15-31
- “Troubleshoot Back to Model” on page 15-34

Install Polyspace Plugin for Simulink

By default, when you install Polyspace R2013b or later, the Simulink plugin is installed and connected to MATLAB.

If you model on a previous version of Simulink and MATLAB, you can also connect the Polyspace plugin on this previous version. That way you use the latest analysis software with your preferred version of Embedded Coder or TargetLink®. The Simulink plugin supports the four previous releases of MATLAB. For example, the R201b version of the Polyspace plugin supports MATLAB versions R2015b through R2017b.

If you use a cross-version of Polyspace and MATLAB, local batch analyses can only be submitted from the Polyspace environment or using the `pslinkrun` command.

Note To install a newer version of Polyspace on MATLAB R2013b or later, you must install MATLAB without the corresponding version of Polyspace.

---

1. Using an account with read/write privileges, open the older version of MATLAB.
2. Use the `ver` command to make sure you do not have a previous version of Polyspace installed. See preceding note.
3. Change your **Current Folder** to

   matlabroot`\toolbox\polyspace\pslink\pslink`

   matlabroot is the version of Polyspace you want to connect, for example, `C:\Program Files\MATLAB\R2017b`.
4. Connect the new version of Polyspace by running the command `pslinksetup('install')`.

See Also

Related Examples

- "Verify Code from a Simple Simulink Model"

More About

- "Troubleshoot Back to Model" on page 15-34

Verify Generated Code Using Polyspace Code Prover

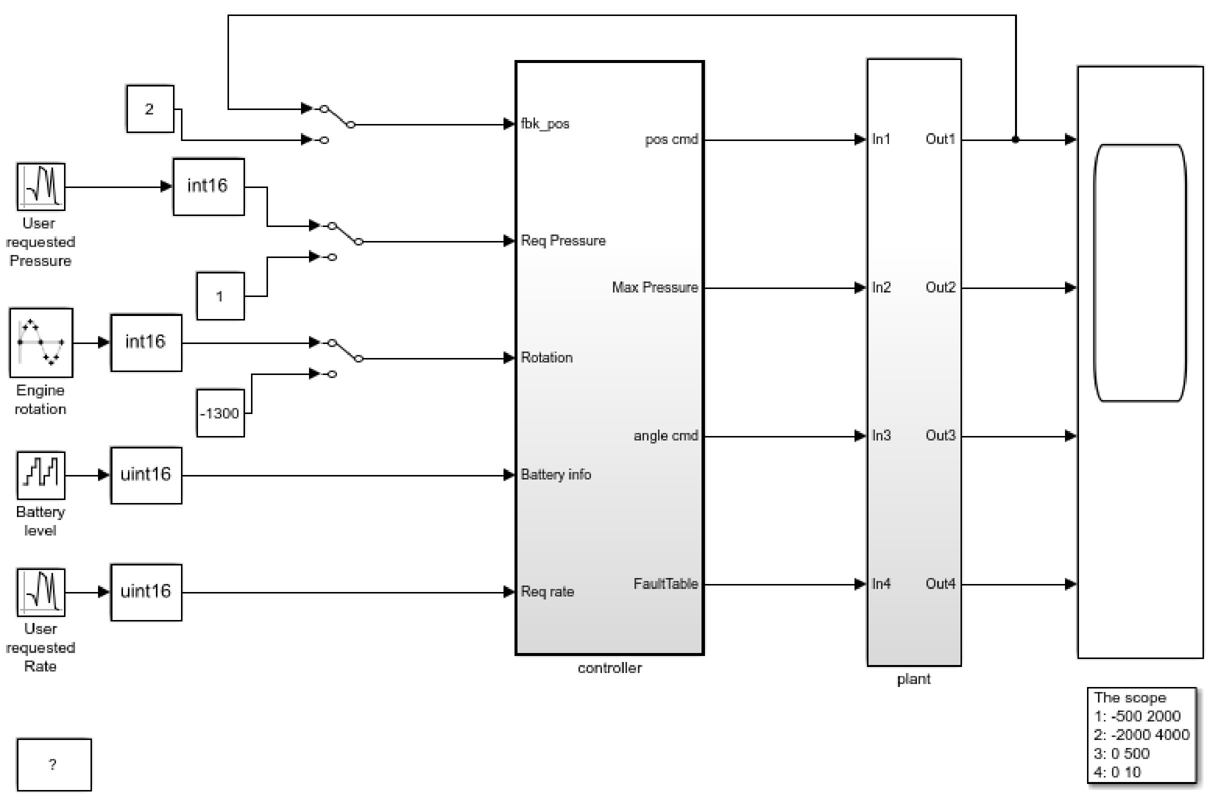
If you generate C or C++ code from models using Embedded Coder, you can check the generated code for run-time errors. Polyspace Code Prover proves code correctness, finds run-time errors, and checks for MISRA-C compliance in generated and handwritten code.

This example contains a demo model from which you can generate code and then analyze the generated code.

Open Model

Open and explore the example model. The model contains a `controller` subsystem, which itself contains many subsystems. One of the subsystems has some issues that can lead to run-time errors in the generated code.

```
open_system('psdemo_model_link_sl');
```



Generate and Analyze Code

Generate code from the `controller` subsytem or one of the subsystems underneath. Then, run Polyspace Code Prover on the generated code. You can trace back from run-time errors found in the generated code to corresponding blocks in the model. You can also check for coding rule violations and add annotations on blocks to justify the violations. For details, see "Verify Code Generated from Simulink Subsystem" on page 15-7.

The `controller` subsystem also contains an S-Function block. You can separately analyze the C code that the S-Function block refers to. For details, see "Verify S-Function Code" on page 17-7.

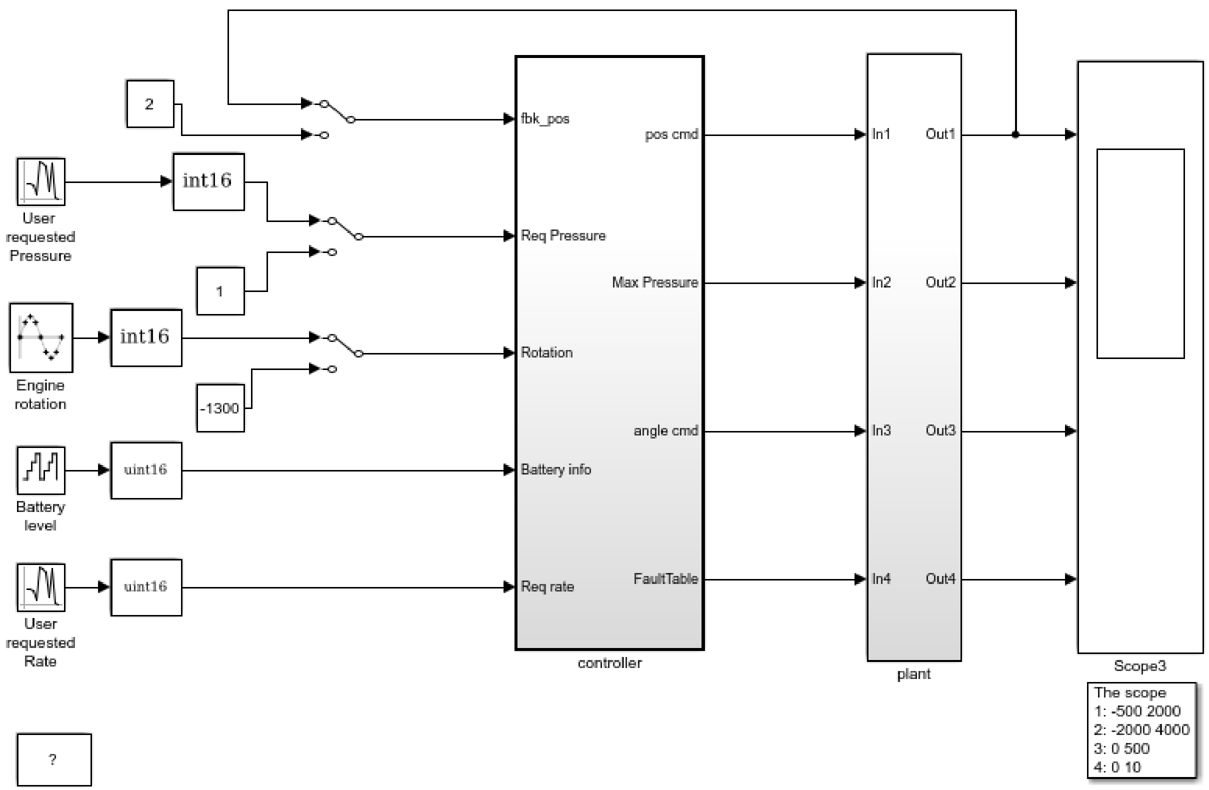
Verify Code Generated from Simulink Subsystem

Polyspace Code Prover proves code correctness, finds run-time errors, and checks for MISRA-C compliance in generated and handwritten code. Use the following workflow for verifying generated code.

Open Model

Open the example model.

```
modelName = 'psdemo_model_link_sl';
open_system(modelName)
```



Generate Code

Generate code for the `controller` subsystem in your model.

1 Right-click the `controller` subsystem and select **C/C++ Code** > **Build This Subsystem**.
2 In the dialog box, select **Build**.

Equivalent MATLAB Code:

```
subsysPath = 'psdemo_model_link_sl/controller';
rtwbuild(subsysPath);
```

Verify Code

Verify the code generated for the `controller` subsystem.

1. Choose Polyspace Code Prover to analyze the code.

   Right-click the `controller` subsystem and select **Polyspace** > **Options**. For **Product mode**, choose `Code Prover`.
2. Analyze the generated code.

   Right-click the `controller` subsystem and select **Polyspace** > **Verify Generated Code for** > **Selected Subsystem**. Follow the progress of verification in the MATLAB Command Window.

Equivalent MATLAB Code:

```
opts = polyspace.ModelLinkOptions('C');
mlopts = pslinkoptions(subsysPath);
mlopts.VerificationMode = 'CodeProver';
mlopts.PrjConfigFile = generateProject(opts, 'polyspaceProject');
pslinkrun(subsysPath, mlopts);
```

For more information on the code, see `polyspace.ModelLinkOptions`, `pslinkoptions` and `pslinkrun`.

Review Verification Results

After verification, the results are displayed in the Polyspace user interface. 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 verification result in detail. For instance:

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 `controller` subsystem.

Trace Errors Back to Model and Fix Them

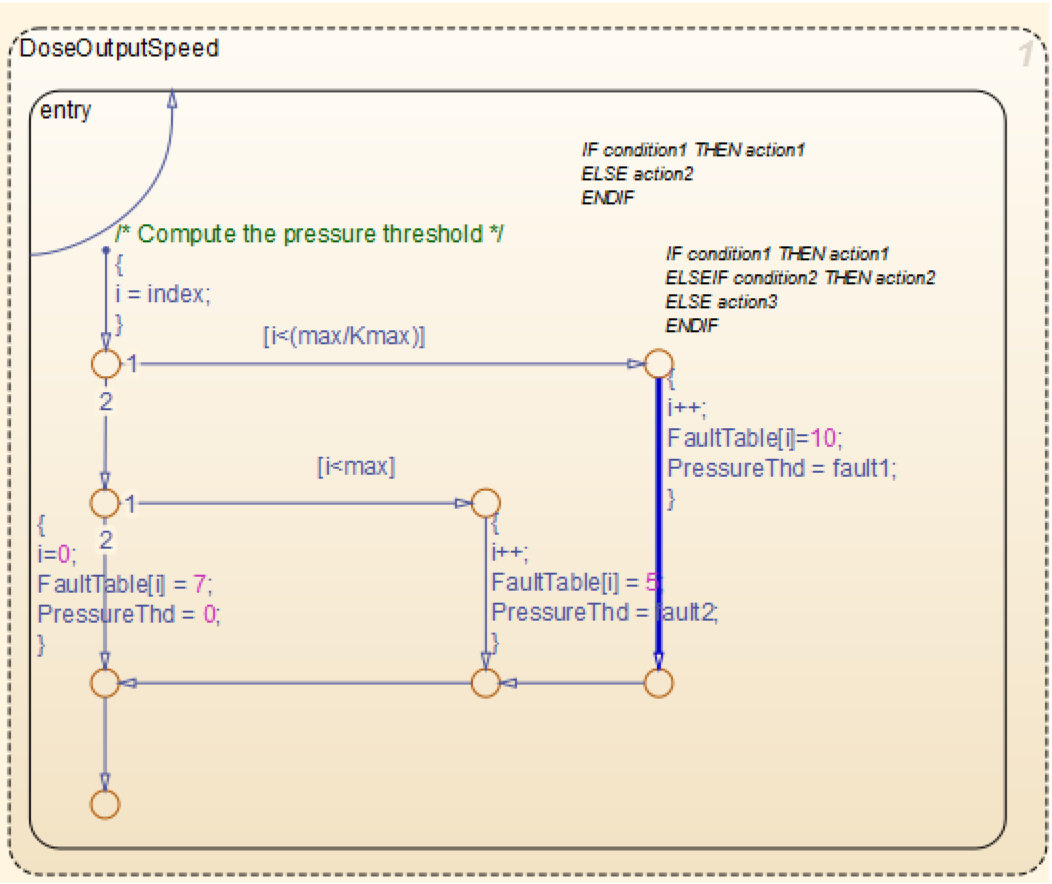
For code generated from the model, you can trace an error back to your model.

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 `controller.c`.
2. On the **Source** pane, click the link **S5:76** in comments above the orange error.

```
/* Transition: '<S5>:75' */
/* Transition: '<S5>:76' */
(*i)++;

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

You see that the error occurs due to a transition in the Stateflow chart `synch_and_asynch_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.

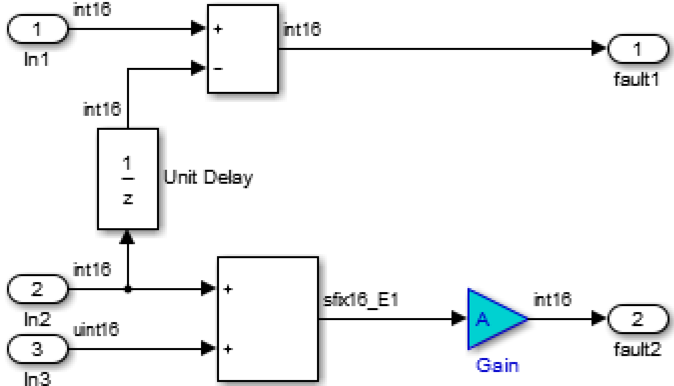
Error 2: Overflow

1. On the **Results List** pane, select the orange **Overflow** error shown below. The error appears in the file `controller.c`.
2. On the **Source** pane, review the error. To trace the error back to the model, click the link **S2/Gain** in comments above the orange error.

```
/* Gain: '<S2>/Gain' incorporates:
* Inport: '<Root>/Battery Info'
```

```
* Inport: '<Root>/Rotation'
* Sum: '<S2>/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 `controller` subsystem.
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 verification:

1. Right-click the `controller` subsystem and select **Polyspace** > **Options**.
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`.
3. Click **Apply** or **OK** and rerun the verification.

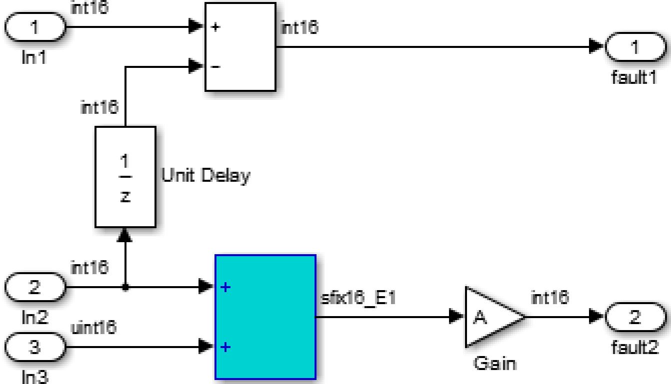
Annotate Blocks to Justify Results

You can justify your results by adding annotations to your blocks. During code verification, 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 `controller.c`, in the function `controller_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 **S2/Sum1** in comments above the addition operation.

   ```
   /* Gain: '<S2>/Gain' incorporates:
   * Inport: '<Root>/Battery Info'
   * Inport: '<Root>/Rotation'
   * Sum: '<S2>/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 Right-click the block and select **Polyspace** > **Annotate Selected Block** > **Edit**.

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 **Not a defect**.

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 verification. The **Severity** and **Status** columns on the **Results List** pane are prepopulated with your annotations.

See Also

More About

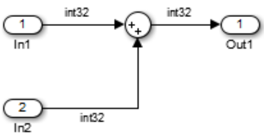
- “Polyspace Configuration for Generated Code” on page 16-2
- “Run Analysis for Embedded Coder” on page 17-3
- “Run Analysis for TargetLink” on page 17-5
- “Configure Model for Code Analysis”
- “Recommended Model Settings for Code Analysis” on page 14-3

- "Troubleshoot Back to Model" on page 15-34

Auto-Annotate Generated Code to Justify Checks

With the Polyspace Code Prover product you can apply Polyspace verification to Embedded Coder generated code. The software detects run-time errors in the generated code and helps you to locate and fix model faults.

Polyspace might highlight overflows for certain operations that are legitimate because of the way the code generator implements these operations. Consider the following model and the corresponding generated code.




```
32 /* Sum: '<Root>/Sum' incorporates:
33   *  Inport: '<Root>/In1'
34   *  Inport: '<Root>/In2'
35   */
36  qY_0 = sat_add_U.In1 + sat_add_U.In2;
37  if ((sat_add_U.In1 < 0) && ((sat_add_U.In2 < 0) && (qY_0 >= 0))) {
38    qY_0 = MIN_int32_T;
39  } else {
40   if ((sat_add_U.In1 > 0) && ((sat_add_U.In2 > 0) && (qY_0 <= 0))) {
41      qY_0 = MAX_int32_T;
42    }
43  }
```

The code generator recognizes that the largest built-in data type is 32-bit. It is not possible to saturate the results of the additions and subtractions using `MIN_INT32` and `MAX_INT32` and a bigger single-word integer data type. Instead the software detects the results overflow and the direction of the overflow, and saturates the result.

If you do not provide justification for the addition operator on line 36, Polyspace verification generates an orange check that indicates a potential overflow. The verification does not take into account the saturation function of lines 37 to 43. In addition, the trace-back functionality of Polyspace Code Prover does not identify the reason for the orange check.

To justify overflows from operators that are legitimate, on the **Configuration Parameters** > **Code Generation** > **Comments** pane:

- Under **Overall control**, select the **Include comments** check box.
- Under **Auto generate comments**, select the **Operator annotations** check box.

The code generator annotates generated code with comments for Polyspace. For example:

```
32 /* Sum: '<Root>/Sum' incorporates:
33   *  Inport: '<Root>/In1'
34   *  Inport: '<Root>/In2'
35   */
36  qY_0 = sat_add_U.In1 +/*MW:OvOk*/ sat_add_U.In2;
```

When you run a verification using Polyspace Code Prover, the Polyspace software uses the annotations to justify the operator-related orange checks and assigns the `Not a defect` classification to the checks.

See Also

Related Examples

- “Annotate Blocks for Known Results” on page 14-12

Main Generation for Model Verification

When you run a verification, the software automatically reads the following information from the model:

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

The software then uses this information to generate a `main` function that:

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

If the `codeInfo` for the model does not contain the names of the inputs, the software considers all variables as entries, except for parameters and outputs.

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

- For class methods in the generated code, the variables that are written before and in the loop refer to the class members.
- For functions with global scope, the associated variables are also in the global scope.

Example 15.1. `main` for Generated Code

The following example shows the `main` generator options that the software uses to generate the `main` function for code generated from a Simulink model.

```
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
```

Configure Data Range Settings

There are two approaches to code verification, which can produce results that are slightly different:

- **Contextual Verification** — Prove code does not generate run-time errors under predefined working conditions. This limits the scope of the verification to specific variable ranges, and verifies the code within these ranges.
- **Robustness Verification —** Prove code generate run-time errors for all verification conditions, including “abnormal” conditions for which the code was not designed. This can be thought of as “worst case” verification.

You perform contextual or robustness verification by the way you specify data ranges for model inputs, outputs, and tunable parameters within the model.

To specify data range settings for your model:

1. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.
2. In the Data Range Management section, specify how you want the verification to treat:

   a. **Input** — Select one of the following:

      - `Use specified minimum and maximum values` (Default) — Apply data ranges defined in blocks or base workspace to increase the precision of the verification. See “Specify Signal Ranges” on page 16-15.
      - `Unbounded inputs` — Assume all inputs are full-range values (`min...max`)

   b. **Tunable parameters** — Select one of the following:

      - `Use calibration data` (Default) — Use value of constant parameter specified in code.
      - `Use specified minimum and maximum values` — Use a parameter range defined in the block or base workspace. See “Specify Signal Ranges” on page 16-15. If no range is defined, use full range (`min...max`).

   c. **Output** — Select one of the following:

      - `No verification` (Default) — No assertion ranges on outputs.

- `Verify outputs are within minimum and maximum values` — Use assertion ranges on outputs.

Note This mode is incompatible with the Automatic Orange Tester.

In general, you should use the following combinations:

- To maximize verification precision, select `Use specified minimum and maximum values` for **Input** and **Tunable parameters**.
- To verify the extreme cases of program execution, select `Unbounded inputs` for **Input** and `Use calibration data` for **Tunable parameters**.

See Also

Related Examples

- "Specify Signal Ranges" on page 16-15

Embedded Coder Considerations

In this section...
“Default Options” on page 15-22
“Data Range Specification” on page 15-22
“Recommended Polyspace options for Verifying Generated Code” on page 15-23
“Hardware Mapping Between Simulink and Polyspace” on page 15-26

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.

Data Range Specification

You can constrain inputs, parameters, and outputs to lie within specified data ranges for Embedded Coder and AUTOSAR with Embedded Coder. See “Configure Data Range Settings” on page 15-20.

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

You can also manually define a DRS file in the Polyspace user interface. If you define a DRS file, the software appends the automatically generated information to the DRS file you create. Manually defined DRS information overrides automatically generated information for all variables.

The software supports the automatic generation of data range 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

The software supports the automatic generation of data range specifications for only the following signal and parameter storage classes:

- `SimulinkGlobal`
- `ExportedGlobal`
- `Struct (Custom)`

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. To open this pane:

1. In the Simulink model window, select **Code** > **Polyspace** > **Options** . The **Polyspace Model Link** pane opens.
2. Click **Configure**. The Polyspace user interface opens, displaying the Polyspace **Configuration** pane.

The following table describes options that you should specify in your Polyspace project before verifying code generated by Embedded Coder software.

Option	Recommended Value	Comments
Macros > Preprocessor definitions `-D`	See Comments	Defines macro compiler flags used during compilation. Use one `-D` for each line of the Embedded Coder generated `defines.txt` file. Polyspace Model Link™ SL does not do this by default.
Environment Settings > Code from DOS or Windows file system `-dos`	On	You must select this option if the contents of the include or source directory comes from a DOS or Windows file system. The option allows the verification to deal with upper/lower case sensitivity and control characters issues. Concerned files are: • **Header files** – All include folders specified (`-I` option) • **Source files** – All source files selected for the verification (`-sources` option)
Check Behavior > Allow negative operands for left shifts `-allow-negative-operand-in-shift`	On	Allows a shift operation on a negative number. According to the ANSI standard, such a shift operation on a negative number is illegal. For example, -2 << 2 If you select this option, Polyspace considers the operation to be valid. For the given example, -2 << 2 = -8

Option	Recommende d Value	Comments
Precision > Precision level `-O`	`2`	Specifies the precision level for the verification. Higher precision levels provide higher selectivity at the expense of longer verification time. Begin with the lowest precision level. You can then address red errors and gray code before rerunning the Polyspace verification using higher precision levels. Benefits: A higher precision level contributes to a higher selectivity rate, making results review more efficient and hence making bugs in the code easier to isolate. The precision level specifies the algorithms used to model the program state space during verification: • `-O0` corresponds to static interval verification. • `-O1` corresponds to complex polyhedron model of domain values. • `-O2` corresponds to more complex algorithms to closely model domain values (a mixed approach with integer lattices and complex polyhedrons). • `-O3` is suitable only for units smaller than 1,000 lines of code. For such code, selectivity may reach as high as 98%, but verification may take up to an hour per 1,000 lines of code.

Option	Recommende d Value	Comments
Precision > Verification level `-to`	See comments	Specifies the phase after which the verification stops. **`Source compliance checking`** – When checking coding rule compliance only. **`Software safelty analysis level 0`** – When verifying code for the first time. **`Software safelty analysis level 4`** – When performing subsequent verifications of code. Each verification phase improves the selectivity of your results, but increases the overall verification time. Improved selectivity can make results review more efficient, and hence make bugs in the code easier to isolate. Begin by running `-to pass0` (`Software Safety Analysis level 0`) You can then address red errors and gray code before relaunching verification using higher integration levels.

Hardware Mapping Between Simulink and Polyspace

The software automatically imports target word lengths and byte ordering (endianess) 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.

TargetLink Considerations

In this section...

"TargetLink Support" on page 15-27
"Default Options" on page 15-27
"Lookup Tables" on page 15-28
"Data Range Specification" on page 15-28
"Code Generation Options" on page 15-29
"Justifying Results in Model" on page 15-29

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]
-ignore-constant-overflows
-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 “Specify Signal Ranges” on page 16-15.

The software automatically creates a Polyspace constraints file using the dSPACE Data Dictionary for each global variable. The DRS 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 DRS file in the Polyspace user interface. If you define a DRS file, the software appends the automatically generated information to the DRS file you create. Manually defined DRS information overrides automatically generated information for all variables.

DRS 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).

Justifying Results in Model

You cannot annotate blocks to justify results in code generated using TargetLink.

This feature is supported only for code generated using Embedded Coder.

See Also

Related Examples

- "Run Analysis for TargetLink" on page 17-5

External Websites

- dSPACE – TargetLink

View Results in Polyspace Code Prover

When a verification completes, you can view the results in the Polyspace user interface.

To view your results:

1. From the Simulink model window, select **Code** > **Polyspace** > **Open Results**.

   Note If you set **Model reference verification depth** to `All` and selected **Model by model verification**, the Select the Result Folder to Open in Polyspace dialog box opens. The dialog box displays a hierarchy of referenced models from which the software generates code. To view the verification results for code generated from a specific model, select the model from the hierarchy. Then click **OK**.

   You can also open results through a Model block or subsystem. From the Simulink model window, right-click the Model block or subsystem, and from the context menu, select **Polyspace** > **Open Results**.

   After a few seconds, the Polyspace user interface opens.
2. On the **Results List** tab, click any check to review additional information.

   The **Result Details** pane shows information about the orange check, and the **Source** pane shows the source code containing the orange check.

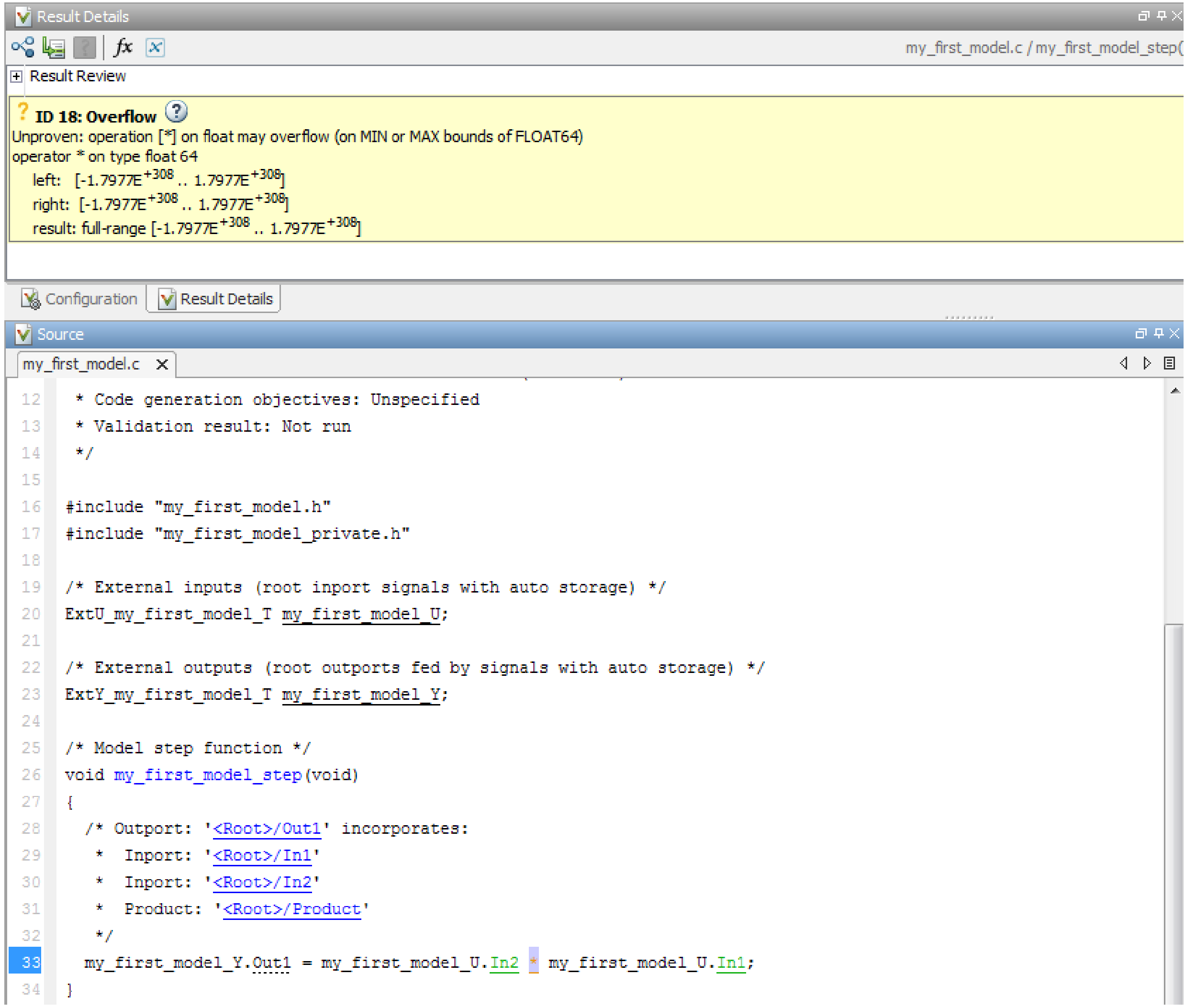
For more information on reviewing run-time checks, see “Review Analysis Results”.

For information on specific checks, see “Run-Time Checks”.

Identify Errors in Simulink Models

With Polyspace Code Prover, you can trace run-time checks in your verification results directly to your Simulink model.

Consider the following example, where the **Result Details** pane shows information about an orange check, and the **Source** pane shows the source code containing the orange check.



This orange check shows a potential overflow issue when multiplying the signals from the inports `In1` and `In2`. To fix this issue, you must return to the model.

To trace this run-time check to the model:

1. Click the blue underlined link (`<Root>/Product`) immediately before the check in the **Source** pane. The Simulink model opens, highlighting the block with the error.

2. Examine the model to find the cause of the check.

   In this example, the highlighted block multiplies two full-range signals, which could result in an overflow. This could be a flaw in:

   - Design — If the model is supposed to be robust for the full signal range, then the issue is a design flaw. In this case, you must change the model to accommodate the full signal range. For example, you could saturate the output of the previous block, or bound the signal with a Switch block.
   - Specifications — If the model is supposed to work for specific input ranges, you can provide these ranges using block parameters or the base workspace. The verification will then read these ranges from the model. See “Specify Signal Ranges” on page 16-15.

   Applying either solution should address the issue and cause the orange check to turn green.
3. To investigate a check further, sometimes you have to trace an instance of a variable in generated code back to your model.

   Right-click an identifier and select **Go To Model**. The model shows the corresponding block highlighted in blue. If the block is in a subsystem, both the subsystem and the block are highlighted in blue.

See Also

More About

- “Troubleshoot Back to Model” on page 15-34

Troubleshoot Back to Model

In this section...
"Back-to-Model Links Do Not Work" on page 15-34
"Your Model Already Uses Highlighting" on page 15-34

Back-to-Model Links 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'`

Configure Code Analysis Options

- “Polyspace Configuration for Generated Code” on page 16-2
- “Include Handwritten Code” on page 16-3
- “Configure Analysis Depth for Referenced Models” on page 16-4
- “Configure Advanced Polyspace Analysis Options” on page 16-5
- “Set Custom Target Settings” on page 16-8
- “Set Up Remote Analysis” on page 16-11
- “Manage Results” on page 16-12
- “Specify Signal Ranges” on page 16-15

Polyspace Configuration for Generated Code

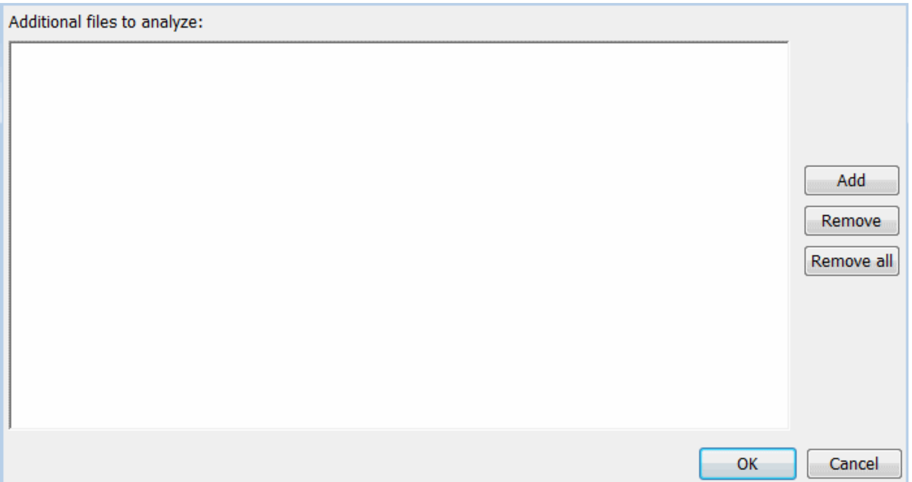
You do not have to manually create a Polyspace project or specify Polyspace options before running an analysis for your generated code. By default, Polyspace automatically creates a project and extracts the required information from your model. You can modify this configuration and or specify additional options for your analysis with the Polyspace configuration options:

- You may incorporate separately created code within the code generated from your Simulink model. See “Include Handwritten Code” on page 16-3.
- By default, the Polyspace analysis is contextual and treats tunable parameters as constants. You can specify a verification that considers robustness, including tunable parameters that lie within a range of values. See “Configure Data Range Settings” on page 15-20.
- You may customize the options for your analysis. For example, to specify the target environment or adjust precision settings. See “Configure Advanced Polyspace Analysis Options” on page 16-5 and “Recommended Polyspace options for Verifying Generated Code” on page 15-23.
- You may create specific configurations for batch runs. See “Use a Saved Polyspace Configuration File Template” on page 16-6.
- If you want to analyze code generated for a 16-bit target processor, you must specify header files for your 16-bit compiler. See “Set Custom Target Settings” on page 16-8.

Include Handwritten Code

Files such as S-Function wrappers are, by default, not part of the Polyspace analysis. However, you can add these files to your generated code analysis manually. You can also analyze your S-Functions separately on page 17-7.

1. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.
2. Select the **Enable additional file list** check box. Then click **Select files**. The Files Selector dialog box opens.



3. Click **Add**. The Select files to add dialog box opens.
4. Use the Select files to add dialog box to:

   - Navigate to the relevant folder
   - Add the required files.

   The software displays the selected files as a list under **Additional files to analyze**.

   Note To remove a file from the list, select the file and click **Remove**. To remove all files from the list, click **Remove all**.

   ---
5. Click **OK**.

Configure Analysis Depth for Referenced Models

From the **Polyspace** pane, you can specify the analysis of generated code with respect to model reference hierarchy levels:

- **Model reference verification depth** — From the drop-down list, select one of the following:
  - `Current model only` — Default. The Polyspace runs code from the top level only. The software creates stubs to represent code from lower hierarchy levels.
  - `1` — The software analyzes code from the top level and the next level. For subsequent hierarchy levels, the software creates stubs.
  - `2` — The software analyzes code from the top level and the next two hierarchy levels. For subsequent hierarchy levels, the software creates stubs.
  - `3` — The software analyzes code from the top level and the next three hierarchy levels. For subsequent hierarchy levels, the software creates stubs.
  - `All` — The software analyzes code from the top level and all lower hierarchy levels.
- **Model by model verification** — Select this check box if you want the software to analyze code from each model separately.

Note The same configuration settings apply to all referenced models within a top model. The options and parameters are the same whether you open the **Polyspace** pane (**Polyspace** > **Options**) from the toolbar or through the right-click context menu. However, you can run analyses for code generated from specific Model blocks. See “Run Analysis for Embedded Coder” on page 17-3.

Configure Advanced Polyspace Analysis Options

From Simulink, you can specify Polyspace options to change the configuration of the Polyspace analysis. For example, you can specify the processor type and operating system of your target device. For descriptions of options, see “Analysis Options”.

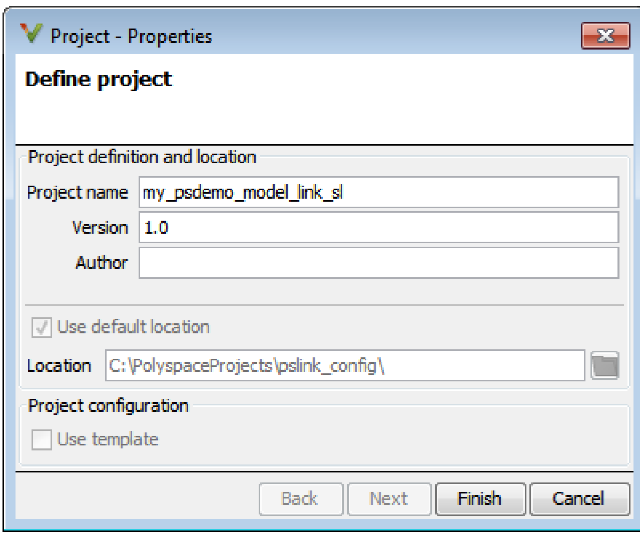
In this section...

“Set Advanced Analysis Options” on page 16-5
“Use a Saved Polyspace Configuration File Template” on page 16-6
“Reset Polyspace Options for a Simulink Model” on page 16-7

Set Advanced Analysis Options

1. From Simulink, select **Code** > **Polyspace** > **Options**.
2. In the Polyspace parameter configuration pane, select **Configure**.
3. In the Polyspace Configuration window that opens, set the options required by your application.

   The first time you open the configuration, the software sets certain options by default depending on your code generator. See Default Embedded Coder Options on page 15-22 or Default TargetLink Options on page 15-27.
4. To change the project name or other project properties, on the toolbar, click the **Project properties** icon



5 Save your changes and close.
6 To use your configuration with other projects, copy the `.psprj` file and rename the updated project configuration file. For example, you can call your cross-compilation configuration `my_cross_compiler.psprj`.

Use a Saved Polyspace Configuration File Template

If you want to reuse a Polyspace configuration for multiple project, you need to add the configuration to the model parameters. This workflow shows how to add a previously created configuration. To create a configuration file template, see "Set Advanced Analysis Options" on page 16-5.

In the Simulink user interface:

1 From Simulink, select **Code** > **Polyspace** > **Options**.
2 In the Polyspace parameter configuration pane, select **Use custom project file**.
3 In the text box, enter the full path to a `.psprj` file, or click **Browse for project file** to browse instead.

At the MATLAB command line:

- Use `pslinkfun('settemplate',...)` to apply a configuration defined by a configuration file template.

For example:

```
pslinkfun('settemplate','C:\Work\my_cross_compiler.psprj')
```

Reset Polyspace Options for a Simulink Model

If you want to reset the Polyspace configuration information to the default, you can remove your custom options from your Simulink model.

1. To remove options from a top model, select **Code** > **Polyspace** > **Remove Options from Current Configuration**.
2. To remove options from a Model block or subsystem, right-click the block or subsystem and select **Polyspace** > **Remove Options from Current Configuration**.
3. Save the model.

See Also

`pslinkfun` | `pslinkoptions`

Related Examples

- "Use a Saved Polyspace Configuration File Template" on page 16-6

More About

- "Embedded Coder Considerations" on page 15-22
- "TargetLink Considerations" on page 15-27
- "Recommended Polyspace options for Verifying Generated Code" on page 15-23

Set Custom Target Settings

If your target has specific setting, you can analyze your code in context of those settings. For example, if you want to analyze code generated for a 16-bit target processor, you must specify header files for your 16-bit compiler. The software automatically identifies the compiler from the Simulink model. If the compiler is 16-bit and you do not specify the relevant header files, the Polyspace will produce an error when you try to run an analysis.

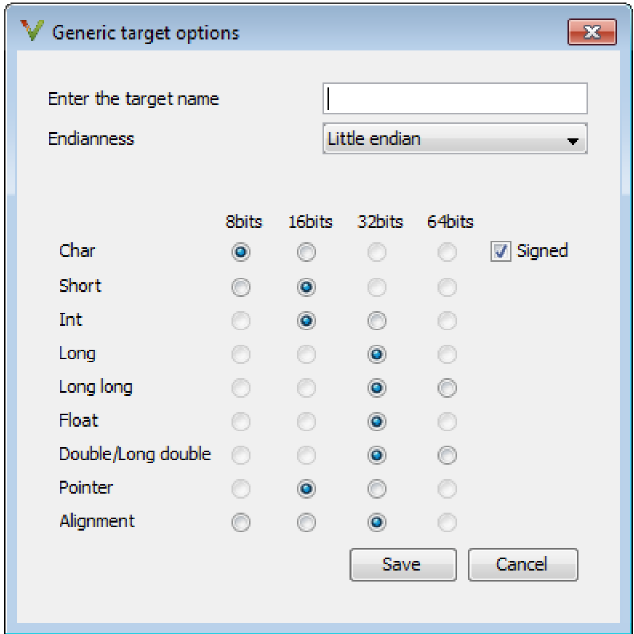
Note For a 32-bit or 64-bit target processor, the software automatically specifies the default header file.

---

1 In the Simulink model window, select **Code** > **Polyspace** > **Options**.
2 Click **Configure**.

   The Polyspace Configuration window opens. Use this pane to customize the target and cross compiler.
3 From the **Configuration** tree, expand the **Target & Compiler** node.
4 In the **Target Environment** section, use the **Target processor type** option to define the size of data types.

   a From the drop-down list, select `mcpu...(Advanced)`. The Generic target options dialog box opens.



Use this dialog box to create a new target and specify data types for the target. Then click **Save**.

5 From the Configuration tree, select **Target & Compiler** > **Macros**. Use the **Preprocessor definitions** section to define preprocessor macros for your cross-compiler.

To add a macro, in the **Macros** table, select ✚. In the new line, enter the required text.

To remove a macro, select the macro and click 🗑.

Note If you use the LCC cross-compiler, then you must specify the `MATLAB_MEX_FILE` macro.

---

6 Select **Target & Compiler** > **Environment Settings**.

7 In the **Include folders** (or **Include**) section, specify a folder (or header file) path by doing one of the following:

- Select and enter the folder or file path.
- Select and use the dialog box to navigate to the required folder (or file).

You can remove an item from the displayed list by selecting the item and then clicking .

8. Save your changes and close.

   To use your configuration settings in other projects, see “Use a Saved Polyspace Configuration File Template” on page 16-6.

Set Up Remote Analysis

By default, the Polyspace software runs locally. To specify a remote analysis:

1. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.
2. Select **Configure**.
3. In the Polyspace Configuration window, select the **Run Settings** pane.
4. Select the **Run Code Prover analysis on a remote cluster** check box.
5. If you use Polyspace Metrics as a results repository, select **Upload results to Polyspace Metrics**.
6. If you have not already connected to a server, from the toolbar, select **Options** > **Preferences**. For help filling in this dialog, see “Configure Polyspace Preferences”.
7. Close the configuration window and apply your changes.

Manage Results

In this section...

"Open Polyspace Results Automatically" on page 16-12
"Specify Location of Results" on page 16-13
"Save Results to a Simulink Project" on page 16-14

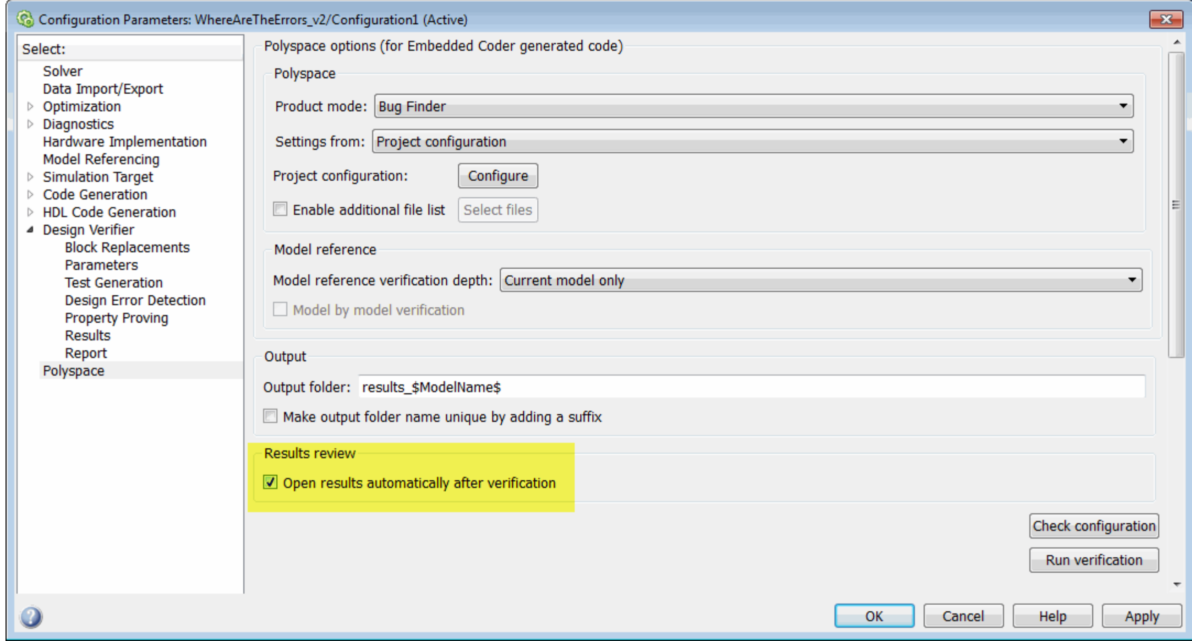
Open Polyspace Results Automatically

You can configure the software to automatically open your Polyspace results after you start the analysis. If you are doing a remote analysis, the Polyspace Metrics web page opens. When the remote job is complete, you can download your results from Polyspace Metrics. If you are doing a local analysis, when the local job is complete, the Polyspace environment opens the results in the Polyspace interface.

To configure the results to open automatically:

1 From the model window, select **Code** > **Polyspace** > **Options**.

   The Polyspace pane opens.



2 In the Results review section, select **Open results automatically after verification**.
3 Click **Apply** to save your settings.

Specify Location of Results

By default, the software stores your results in `Current Folder\results_model_name`. Every time you rerun, your old results are over written. To customize these options:

1 From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens to the Polyspace pane.
2 In the **Output folder** field, specify a full path for your results folder. By default, the software stores results in the current folder.
3 If you want to avoid overwriting results from previous analyses, select **Make output folder name unique by adding a suffix**.

   Instead of overwriting an existing folder, the software specifies a new location for the results folder by appending a unique number to the folder name.

Save Results to a Simulink Project

By default, the software stores your results in `Current Folder\results_model_name`. If you use a Simulink project for your model work, you can store your Polyspace results there as well for better organization. To add your results to a Simulink Project:

1. Open your Simulink project.
2. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens with the Polyspace pane displayed.
3. Select **Add results to current Simulink Project**.
4. Run your analysis.

   Your results are saved to the Simulink project you opened in step 1.

Specify Signal Ranges

If you constrain signals in your Simulink model to lie within specified ranges, Polyspace software automatically applies these constraints during verification of the generated code. Using signal rages can improve the precision of your results.

You can specify a range for a model signal by:

- Applying constraints through source block parameters. See “Specify Signal Range Through Source Block Parameters” on page 16-15.
- Constraining signals through the base workspace. See “Specify Signal Range Through Base Workspace” on page 16-17.

Note You can also manually define data ranges using the constraint setup feature in the Polyspace user interface. If you manually define a constraint specification file, the software automatically appends any signal range information from your model to the constraint specification file. However, manually defined constraint information overrides information generated from the model for all variables.

Specify Signal Range Through Source Block Parameters

You can specify a signal range by applying constraints to source block parameters.

Specifying a range through source block parameters is often easier than creating signal objects in the base workspace, but must be repeated for each source block. For information on using the base workspace, see “Specify Signal Range Through Base Workspace” on page 16-17.

To specify a signal range using source block parameters:

1. Double-click the source block in your model. The Source Block Parameters dialog box opens.
2. Select the **Signal Attributes** tab.
3. Specify the **Minimum** value for the signal, for example, -15.
4. Specify the **Maximum** value for the signal, for example, 15.

Inport

Provide an input port for a subsystem or model.
For Triggered Subsystems, 'Latch input by delaying outside signal' produces the value of the subsystem input at the previous time step.
For Function-Call Subsystems, turning 'On' the 'Latch input for feedback signals of function-call subsystem outputs' prevents the input value to this subsystem from changing during its execution.
The other parameters can be used to explicitly specify the input signal attributes.

Main | Signal Attributes

Output function call

Minimum: -15

Maximum: 15

Data type: Inherit: auto >>

Lock output data type setting against changes by the fixed-point tools

Port dimensions (-1 for inherited): -1

Variable-size signal: Inherit

Sample time (-1 for inherited): -1

Signal type: auto

Sampling mode: auto

OK Cancel Help

5 Click **OK**.

Specify Signal Range Through Base Workspace

You can specify a signal range by creating signal objects in the MATLAB workspace. This information is used to initialize each global variable to the range of valid values, as defined by the min-max information in the workspace.

Note You can also specify a signal range by applying constraints to individual source block parameters. This method can be easier than creating signal objects in the base workspace, but must be repeated for each source block. For more information, see "Specify Signal Range Through Source Block Parameters" on page 16-15.

To specify an input signal range through the base workspace:

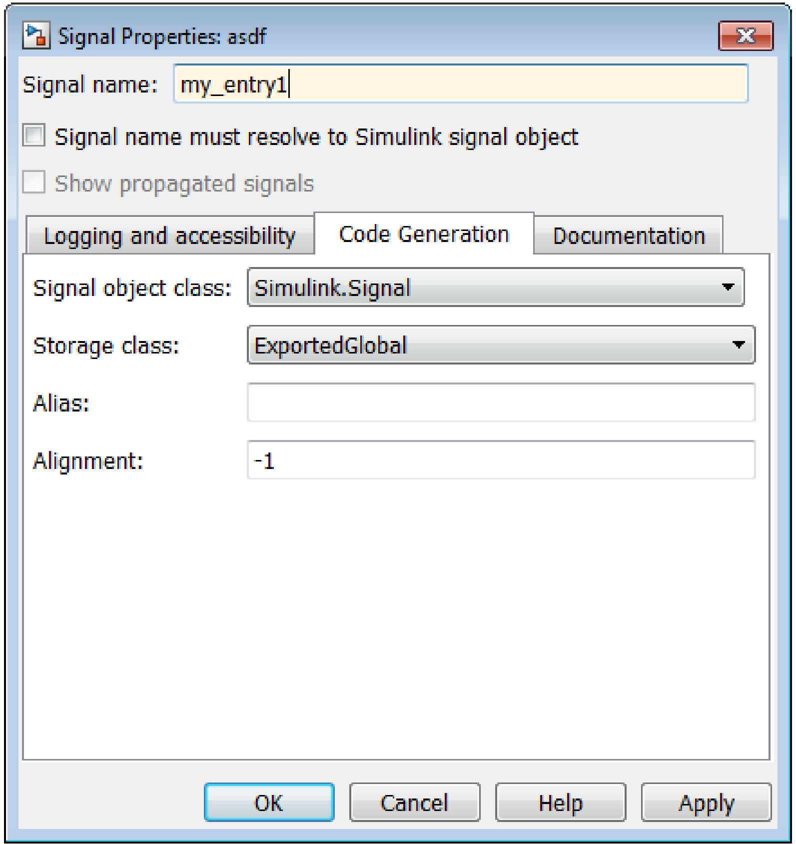
1 Configure the signal to use, for example, the `ExportedGlobal` storage class:

   a Right-click the signal. From the context menu, select **Properties**. The Signal Properties dialog box opens.

   b In the **Signal name** field, enter a name, for example, `my_entry1`.

   c Select the **Code Generation** tab.

   d In the **Storage class** drop-down list, select `ExportedGlobal`.



e. Click **OK**, which applies your changes and closes the dialog box.

2. Using Model Explorer, specify the signal range:

   a. Select **Tools** > **Model Explorer** to open Model Explorer.
   b. From the **Model Hierarchy** tree, select **Base Workspace**.
   c. Create a signal by clicking the `Add Simulink Signal` button. Rename this signal, for example, `my_entry1`.
   d. Set the **Minimum** value for the signal, for example, to `-15`.
   e. Set the **Maximum** value for the signal, for example, to `15`.
   f. From the **Storage class** drop-down list, select `ExportedGlobal`.

g Click **Apply**.

17

Run Polyspace on Generated Code

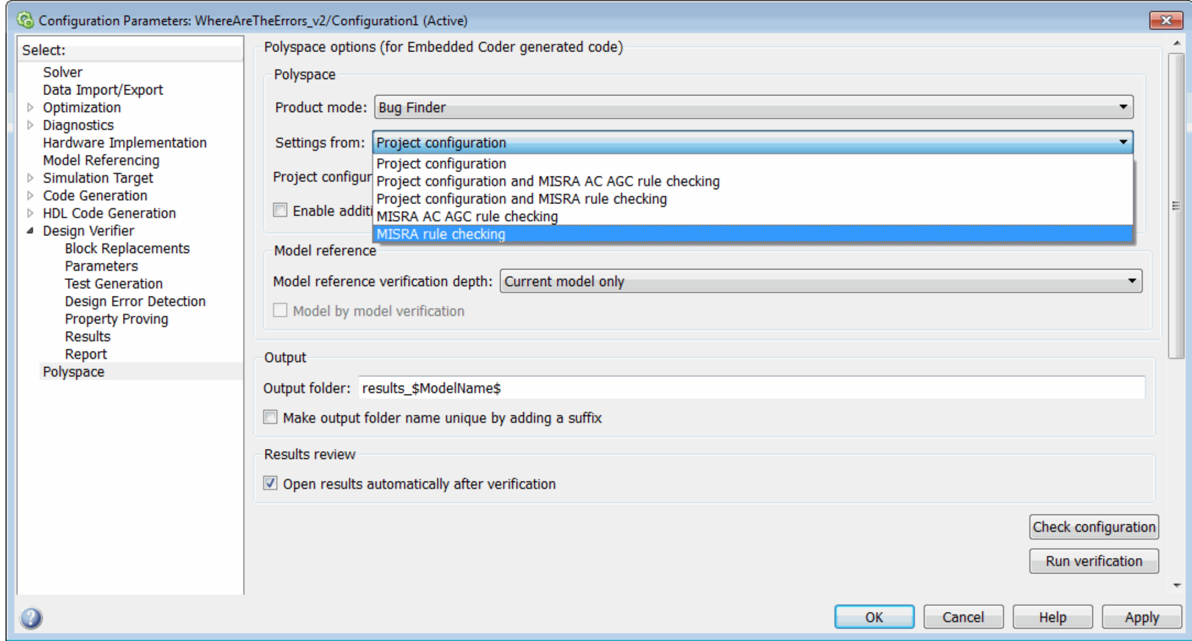
- “Specify Type of Analysis to Perform” on page 17-2
- “Run Analysis for Embedded Coder” on page 17-3
- “Run Analysis for TargetLink” on page 17-5
- “Verify S-Function Code” on page 17-7

Specify Type of Analysis to Perform

Before running Polyspace, you can specify what type of analysis you want to run. You can choose to run code analysis, coding rules checking, or both. You can check compliance with MISRA AC AGC, MISRA C:2004, MISRA C:2012, MISRA C++, and JSF C++ coding rules.

To specify the type of analysis to run:

1. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameter window opens to the **Polyspace** options pane.



2. In the **Settings from** drop-down menu, select the type of analysis you want to perform. For descriptions of the different settings, see “Settings from (C)” or “Settings from (C++)”.

Run Analysis for Embedded Coder

Start the Analysis

There are several different types of analyses you can run on code generated with Embedded Coder. Use the table below to figure out how to start the type of analysis you want.

Code Type to Analyze	What To Select
Code generated from the top model	From the toolbar, select **Code** > **Polyspace** > **Verify Code Generated for** > **Model**.
All code generated as model referenced code	From the toolbar, select **Code** > **Polyspace** > **Verify Code Generated for** > **Referenced Model**.
Model reference code associated with a specific block or subsystem	Right-click the Model block or subsystem and select **Verify Code Generated for** > **Selected Subsystem**

Note You can also start the Polyspace software from the **Polyspace** configuration parameter pane by clicking **Run verification**.

When the Polyspace software starts, messages appear in the MATLAB Command Window:

```
### Starting Polyspace verification for Embedded Coder
### Creating results folder C:\PolySpace_Results\results_my_first_code
                                          for system my_first_code
### Checking Polyspace Model-Link Configuration:
### Parameters used for code verification:
 System                : my_first_code
 Results Folder        : C:\PolySpace_Results\results_my_first_code
 Additional Files      : 0
 Remote                : 0
 Model Reference Depth : Current model only
 Model by Model        : 0
 DRS input mode        : DesignMinMax
 DRS parameter mode    : None
 DRS output mode       : None
...
```

Follow the progress of the analysis in the MATLAB Command Window. If you are running a remote, batch, analysis you can follow the later stages through the Polyspace Job Monitor.

The software writes status messages to a log file in the results folder.

Monitor Progress

Local Analyses

For a local Polyspace runs, you can follow the progress of the software in the MATLAB Command Window. The software also saves the status messages to a log file in the results folder.

Remote Batch Analyses

For a remote analysis, you can follow the initial stages of the analysis in the MATLAB Command Window.

Once the compilation phase is complete, you can follow the progress of the software using the Polyspace Job Monitor.

From Simulink, select **Code** > **Polyspace** > **Open Job Monitor**.

For more information, see “Monitor Progress” on page 6-5.

Run Analysis for TargetLink

To start the Polyspace software on TargetLink generated code:

Start the Analysis

1. In your model, select the Target Link subsystem.
2. In the Simulink model window select **Code** > **Polyspace** > **Verify Code Generated for** > **Selected Target Link Subsystem**.

   Messages appear in the MATLAB Command Window:

```
### Starting Polyspace verification for Embedded Coder
### Creating results folder results_WhereAreTheErrors_v2
                         for system WhereAreTheErrors_v2
### Parameters used for code verification:
    System                : WhereAreTheErrors_v2
    Results Folder        : H:\Desktop\Test_Cases\ModelLink_Testers
                                     \results_WhereAreTheErrors_v2
    Additional Files      : 0
    Verifier settings     : PrjConfig
    DRS input mode        : DesignMinMax
    DRS parameter mode    : None
    DRS output mode       : None
    Model Reference Depth : Current model only
    Model by Model        : 0
```

   The exact messages depend on the code generator you use and the Polyspace product. The software writes status messages to a log file in the results folder.

Follow the progress of the software in the MATLAB Command Window. If you are running a remote, batch analysis, you can follow the later stages in the Polyspace Job Monitor.

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.

Monitor Progress

Local Analyses

For a local Polyspace runs, you can follow the progress of the software in the MATLAB Command Window. The software also saves the status messages to a log file in the results folder.

Remote Batch Analyses

For a remote analysis, you can follow the initial stages of the analysis in the MATLAB Command Window.

Once the compilation phase is complete, you can follow the progress of the software using the Polyspace Job Monitor.

From Simulink, select **Code** > **Polyspace** > **Open Job Monitor**.

For more information, see “Monitor Progress” on page 6-5.

Verify 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.

In this section...
“S-Function Analysis Workflow” on page 17-7
“Compile S-Functions to Be Compatible with Polyspace” on page 17-7
“Example S-Function Analysis” on page 17-8

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 code 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 8-113.
   - 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 SFunctionBuilder block, with **Enable support for Design Verifier** selected on the **Build Info** tab of the SFunctionBuilder dialog box.
- The Simulink Coverage function `slcovmex`, with the option `-sldv`.

Example S-Function Analysis

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

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

```
% Open Model
psdemo_model_link_sl

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

2 Open the subsystem `psdemo_model_link_sl/controller`.

3 Right-click the S-Function block Command_Strategy and select **Polyspace** > **Options**.

4 In the Configuration Parameters dialog box, make sure that the following parameters are set:

   - **Product mode** — `Code Prover`
   - **Settings from** — `Project configuration and MISRA C 2012 checking`
   - **Open results automatically after verification** — ☑ On

5 Apply your settings and close the Configuration Parameters.

6 Right-click the Command_Strategy block and select **Polyspace** > **Verify S-Function** > **This Occurrence**.

7. Follow the analysis in the MATLAB Command Window. When the analysis is finished, your results open in the Polyspace interface.

See Also

Related Examples

- “Include Handwritten Code” on page 16-3
- “Configure Advanced Polyspace Analysis Options” on page 16-5
- “Polyspace Code Prover Results”
- “Configuring S-Function for Test Case Generation” (Simulink Design Verifier)

Using Polyspace Software in the Eclipse IDE

- “Install Polyspace Plugin for Eclipse” on page 18-2
- “Configure Verification” on page 18-5
- “Run Verification” on page 18-9
- “Review Results” on page 18-11

Install Polyspace Plugin for Eclipse

This topic shows how to install or uninstall the Polyspace plugin for Eclipse.

Install Polyspace Plugin for Eclipse IDE

The Polyspace plugin is supported for Eclipse versions 4.3, 4.4, and 4.5. You can install the Polyspace plugin only after you:

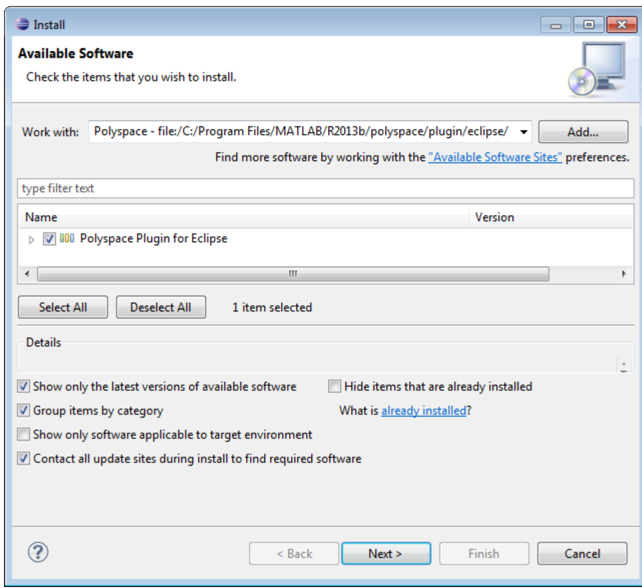
- Install and set up Eclipse Integrated Development Environment (IDE). For more information, see the Eclipse documentation at www.eclipse.org.
- Install Java 7. See Java documentation at www.java.com.

  If you run into issues because of incompatible Java versions, see “Eclipse Java Version Incompatible with Polyspace Plug-in” on page 7-62.
- Uninstall any previous Polyspace plugins. For more information, see “Uninstall Polyspace Plugin for Eclipse IDE” on page 18-4.

To install the Polyspace plugin:

1. From the Eclipse editor, select **Help** > **Install New Software**. The Install wizard opens, displaying the Available Software page.
2. Click **Add** to open the Add Repository dialog box.
3. In the **Name** field, specify a name for your Polyspace site, for example, `Polyspace_Eclipse_PlugIn`.
4. Click **Local**, to open the Browse for Folder dialog box.
5. Navigate to the `MATLAB_Install\polyspace\plugin\eclipse` folder. Then click **OK**.

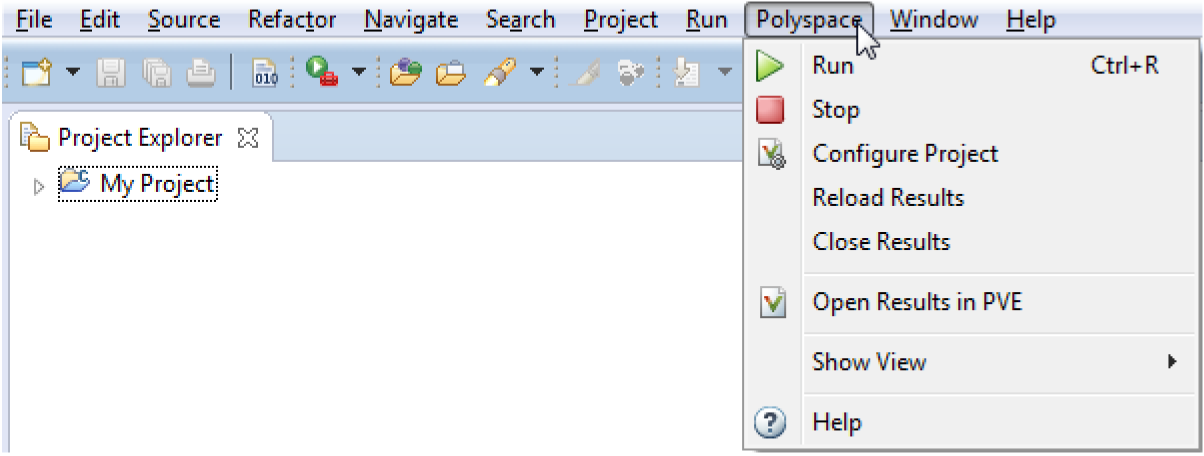
   `MATLAB_Install` is the installation folder for the Polyspace product.
6. Click **OK** to close the Add Repository dialog box.
7. On the Available Software page, select `Polyspace Plugin for Eclipse`.



8 Click **Next**.
9 On the Install Details page, click **Next**.
10 On the Review Licenses page, review and accept the license agreement. Then click **Finish**.

Once you install the plugin, in the Eclipse editor, you'll see:

- A **Polyspace** menu
- A **Polyspace Run - Code Prover**, **Results List - Code Prover**, and **Result Details** view.



Uninstall Polyspace Plugin for Eclipse IDE

Before installing a new Polyspace plugin, you must uninstall any previous Polyspace plugins:

1 In Eclipse, select **Help** > **About Eclipse**.
2 Select **Installation Details**.
3 Select the Polyspace plugin and select **Uninstall**.

   Follow the uninstall wizard to remove the Polyspace plugin. You must restart Eclipse for changes to take effect.

See Also

Related Examples

- “Configure Verification” on page 18-5

More About

- “Analysis in Eclipse”

Configure Verification

This example shows how to set up a Polyspace verification within the Eclipse Integrated Development Environment (IDE). You can run verification on C/C++ code and review the verification results without leaving the Eclipse environment.

Before running verification, you can change the default values for these verification options.

- 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 verification extracts the compiler options from the project. If the project refers to your compilation toolchain through a build command, the verification cannot extract the compiler options. Trace the build command to extract the options.
- Other options: You specify which analysis results you want and how precise you want them to be. If the default values of the options are not optimal for your situation, change them. For instance, the default verification checks for run-time errors only. If you also want coding rule violations, specify the type of rules that you want checked.

Prerequisites

Before you configure your Polyspace verification, you must:

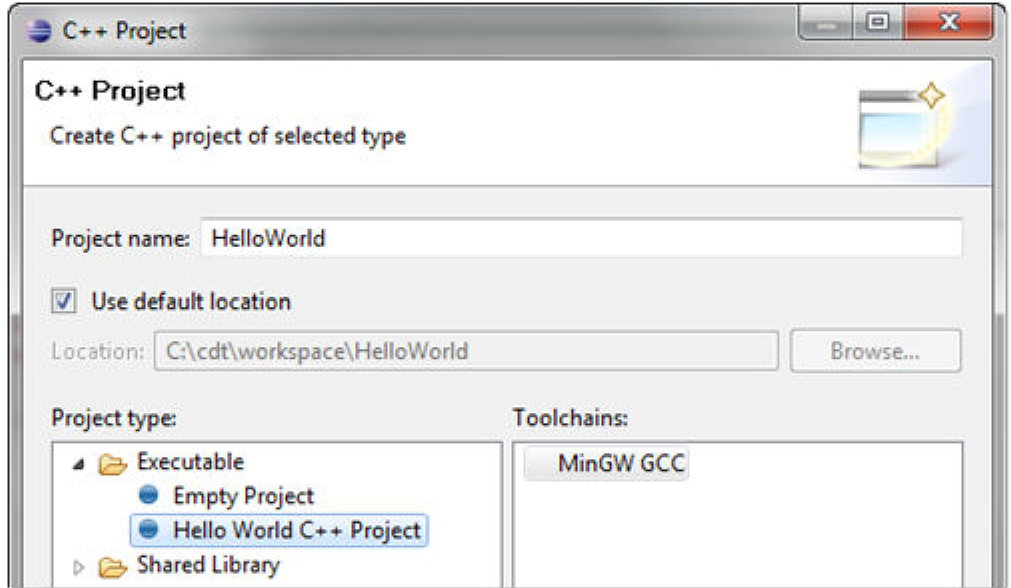
- Install the Polyspace plugin for Eclipse.

  See “Install Polyspace Plugin for Eclipse” on page 18-2.
- Set up an Eclipse project containing the source code that you want to verify.

  See Eclipse documentation.

Eclipse Directly Refers 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. If you directly refer to your compilation toolchain in the Eclipse project, configure the verification as follows.



Compiler Options

The compiler options from your Eclipse project, such as include paths and preprocessor macros, are reused for the verification.

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 verification, the paths are implicitly used with the verification option `-I`.
- See the preprocessor macros under the **Symbols** tab.

  During verification, the macros are implicitly used with the verification option `Preprocessor definitions (-D)`.

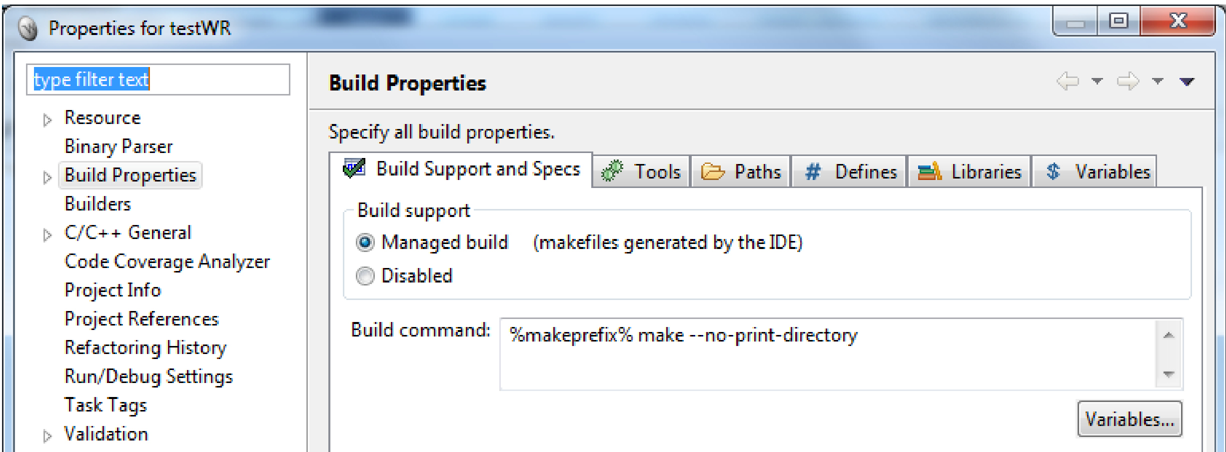
Other Options

You specify other options not related to your compiler directly in the Polyspace configuration. To open the Polyspace Code Prover Configuration window, select

Polyspace > Configure Project. Specify the verification options and save them. For more information, see "Analysis Options".

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 refer to your compilation toolchain through a build command, configure the verification as follows.



Compiler Options

If you use a build command for compilation, the verification cannot automatically extract the compiler options. You must trace your build command.

1 Replace your build command with:

```
matlabroot\polyspace\bin\polyspace-configure.exe
  -output-project polyspaceworkspace\Projects\EclipseProjects\
  projectName\projectName.psprj buildCommand
```

Here:

- *matlabRoot* is the MATLAB 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).
- *projectName* 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
  ```

2 Build your Eclipse project. Perform a clean build so that all 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 verification on your Eclipse project. The verification uses the compiler options that it has extracted.

Other Options

You can specify other options not related to your compiler directly in the Polyspace configuration. Select **Polyspace** > **Configure Project** to open the Polyspace Code Prover Configuration window. Specify the verification options and save them. For more information, see “Analysis Options”.

Next Steps

After you configure your project, you are ready to run verification. See “Run Verification” on page 18-9.

Run Verification

This example shows how to run a Polyspace verification within the Eclipse Integrated Development Environment (IDE).

Prerequisites

Before you run Polyspace verification, you must do the following:

- Install the Polyspace plugin for Eclipse.

  See “Install Polyspace Plugin for Eclipse” on page 18-2.
- Set up an Eclipse project for the source code that you want to verify. Configure Polyspace verification for the project.

  See “Configure Verification” on page 18-5.

Start, Monitor and Stop Verification

You can start a Polyspace verification from the Eclipse editor.

1. Switch to the Polyspace perspective.

   a. Select **Window** > **Open Perspective** > **Other**.

   b. In the Open Perspective dialog box, select **Polyspace**.

   This allows you to view only the information related to a Polyspace verification.
2. If you previously ran a Polyspace Bug Finder analysis, open the **Polyspace Run - Bug Finder** view. In the dropdown list beside the icon, select **Code Prover**.
3. To start a verification, do one of the following:

   - In the **Project Explorer**, right-click the project containing the files that you want to verify and select **Run Polyspace Code Prover**.
   - In the **Project Explorer**, select the project containing the files that you want to verify. From the global menu, select **Polyspace** > **Run**.
4. Follow the progress of the verification in the **Polyspace Run - Code Prover** view.

If you see an error or warning during the compilation phase, double-click it to go to the corresponding location in the source code. Once the verification is over, the results are displayed in the **Results List - Code Prover** view.

5. To stop a verification, select **Polyspace** > **Stop**. Alternatively you can use the button in the **Polyspace Run - Code Prover** view.

The Polyspace files for your Eclipse project, including results and Polyspace configuration files, are saved in the following folder:

`Polyspace_Workspace``\Projects\EclipseProjects\`*`Eclipse Project Name`*

Here:

- *`Eclipse Project Name`* is the name of your Eclipse project.
- *`Polyspace_Workspace`* is the location 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).

If you prefer to store your results within your Eclipse project, inside your Eclipse project folder, create a folder named `polyspace`. Polyspace will save your verification results inside this folder.

Next Steps

After you run a verification in Eclipse, your results open automatically on the **Results List - Code Prover** view. You have to review each result and determine whether to fix your code or justify the result. See "Review Results" on page 18-11.

Review Results

This example shows how to review the results of a Polyspace verification within the Eclipse Integrated Development Environment (IDE).

After you run a verification in Eclipse, your results open automatically on the **Results List - Code Prover** view. You have to review each result and determine whether to fix your code or justify the result.

Prerequisites

To see results from a Polyspace verification, you must do the following:

- Install the Polyspace plugin for Eclipse.

  See "Install Polyspace Plugin for Eclipse" on page 18-2.
- Set up an Eclipse project for the source code that you want to verify. Configure Polyspace verification for the project and run verification.

  See "Run Verification" on page 18-9.

Review Results

After verification, you review each result and determine whether to fix your code or justify the result.

1. Select a result and see detailed information on the **Result Details** view.
2. In the **Result Details** view, to see a brief description and examples of the result, click the button next to the result name.
3. If you close Eclipse or run Polyspace on another Eclipse project, your results are closed. To reopen the results for an Eclipse project, select the project in the **Project Explorer**. From the global menu, select **Polyspace** > **Reload Results**.

Save Multiple Results

The results in Eclipse are overwritten every time a new verification is performed. However, Polyspace automatically imports **Status**, **Severity**, and **Comment** information to the new verification results.

If you want to save your earlier results:

1. Select **Polyspace** > **Open Results in PVE** to open the results in the Polyspace user interface.
2. Save your results from the Polyspace user interface.

   If you have setup Polyspace Metrics, you can upload your results to the web dashboard. For more information, see “Generate Code Quality Metrics” on page 13-12.

In addition to the **Results List** and **Result Details** views available in Eclipse, in the Polyspace user interface, you can use other views to:

- View tooltips with information about variable ranges.
- Navigate the call hierarchy easily in your source code.

Limit Display of Results

This example shows how to control the number and type of results displayed in Eclipse on the results list. To reduce your review effort, you can limit the number of results to display for certain checks or suppress them altogether.

To prevent the analysis from looking for some results, see “Choose Specific Defects” (Polyspace Bug Finder).

If you do not want to change your analysis configuration, you can still change which results are displayed. There are two ways to filter results:

- Filter individual results from display after each run.

  For more information, see “Filter and Group Results” on page 8-113.
- Create a set of filters that you can apply in one click, called a scope.

This example shows how to create a scope:

1. Select **Polyspace** > **Configure Project**.
2. On the configuration window, select **Tools** > **Preferences**.
3. On the **Review Scope** tab, create your filter file.

   a. Select **New**. Save your filter file.

b On the left pane, select a type of result, then use the right pane to select which results to suppress or partially suppress. To suppress a result completely, clear the check box. To partially suppress a result, specify a percentage less than 100 to display.

   If you do not want to use percentages, clear the box **Specify percentage of checks**. This option allows you to specify a specific number of results or the string `ALL`.

   To suppress all results belonging to a category such as **Numerical**, clear the box next to the category name. If only a fraction of results in a category are selected, the check box next to the category name displays a ■ symbol.

4. Select **Apply** or **OK**.

   On the **Results List - Code Prover** pane, a menu displays the list of review scopes.

5. Select the option corresponding to the filters that you want. Only the number or percentage of results that you specify remain on the **Results List - Code Prover** pane.

   - If you specify an absolute number, Polyspace displays that number of results.
   - If you specify a percentage, Polyspace displays that percentage of the total number of results.

See Also

Related Examples

- “Review Red Checks” on page 8-10
- “Review Gray Checks” on page 8-15
- “Review Orange Checks” on page 8-17
- “Review Code Metrics” on page 8-24

Atomic	In computer programming, atomic describes a unitary action or object that is essentially indivisible, unchangeable, whole, and irreducible.
Atomicity	In a transaction involving two or more discrete pieces of information, either all of the pieces are committed or no pieces are committed.
Batch mode	Execution of verification from the command line, rather than via the launcher Graphical User Interface.
Category	One of four types of orange check: *potential bug, inconclusive check, data set issue* and *basic imprecision*.
Certain error	See ”red check.”
Check	A test performed during a verification and subsequently colored red, orange, green or gray in the viewer.
Code verification	The Polyspace process through which code is tested to reveal definite and potential runtime errors and a set of results is generated for review.
Dead Code	Code which is inaccessible at execution time under all circumstances due to the logic of the software executed prior to it.
Development Process	The process used within a company to progress through the software development lifecycle.
Green check	Code has been proven to be free of runtime errors.
Gray check	Unreachable code; dead code.
Imprecision	Approximations are made during a verification, so data values possible at execution time are represented by supersets including those values.
mcpu	Micro Controller/Processor Unit

Orange check	A warning that represents a possible error which may be revealed upon further investigation.
Polyspace Approach	The manner of using verification to achieve a particular goal, with reference to a collection of techniques and guiding principles.
Precision	An verification which includes few inconclusive orange checks is said to be precise
Progress text	Output during verification to indicate what proportion of the verification has been completed. Could be considered as a “textual progress bar”.
Red check	Code has been proven to contain definite runtime errors (every execution will result in an error).
Review	Inspection of the results produced by Polyspace verification.
Scaling option	Option applied when an application submitted for verification proves to be bigger or more complex than is practical.
Selectivitiy	The ratio (green checks + gray checks + red checks) / (total amount of checks)
Unreachable code	Dead code.
Verification	The Polyspace process through which code is tested to reveal definite and potential runtime errors and a set of results is generated for review.